



SocketWrench12

Developer's Guide and Technical Reference
Version 12.0.3024.1286

SocketWrench Licensing Information

The SocketWrench License Agreement provides you with a single developer license and the right to redistribute the .NET assemblies, ActiveX components (OCXs) and libraries (DLLs) included with this product without any additional royalties or runtime licensing fees.

Evaluation Licenses

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

Runtime Licensing

Many languages which use components automatically manage the licensing requirements for those components, and it does not require any additional coding on the part of the developer. For example, placing an ActiveX control on a form in Visual Basic will cause the control's license information to automatically be embedded in the executable. However, in some cases it may be necessary for you to explicitly initialize the control by calling its Initialize method and passing a runtime license key. For example, if an instance of the control is created using the CreateObject function or through a reference, you will need to explicitly initialize it.

When you install SocketWrench with a serial number, a runtime license key will be automatically generated for you and stored in a file in the Include folder where you've installed the product. These files define the SocketWrench runtime licensing key which must be passed to the Initialize method in the components that you are using. If you are using a language that does not have a license key already defined for it, you can create a text file that contains the license key using the License Manager utility. More information about that utility is provided below.

The runtime license key is a null-terminated string that is unique to your licensed copy of SocketWrench. This key should only be embedded in your program and should not be redistributed with your application. If you provide source code for your product, you cannot include the key with the source code.

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

License Manager

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

SocketWrench 12 Upgrade Information

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

Supported Platforms

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

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

Development Tools

The SocketWrench ActiveX control may be used with any programming language that supports the Component Object Model (COM) and ActiveX control interface. This includes languages such as Visual Basic 6.0, Visual FoxPro and PowerBuilder.

The SocketWrench .NET components may be used with Visual Studio 2010 and later versions. If you are developing using Visual Studio 2010, use the .NET 4.0 assemblies. For Visual Studio 2012 and later versions, you can use the .NET 4.5 assemblies. If you are targeting a later version of the Framework, such as .NET 4.7.2 or .NET 4.8, you should reference the .NET 4.5 assemblies, which are compatible with those versions.

If you have updated your application to use .NET with Visual Studio 2022, we recommend referencing the .NET 8.0 LTS assemblies for most projects. SocketWrench 12 also includes assemblies for .NET 9.0 which is a Standard Term Support (STS) release and will only be supported by Microsoft for 18 months.

If you are developing on Windows 7 or Windows 8.1, it is required you use Visual Studio 2010 or a later version. Attempting to use earlier versions of Visual Studio may require that you use those development tools with elevated privileges and they are no longer supported. For Windows 10 and Windows 11, we recommend using Visual Studio 2019 or a later version.

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

Microsoft Visual Studio

The ActiveX controls and dynamic link library may be used with languages such as Visual Basic .NET, Visual C# and Delphi .NET. However, it is recommended that you use the SocketWrench .NET component if you are creating applications that target the .NET Framework. This is particularly true in the case of the ActiveX control which imposes a significant amount of overhead due to how COM interop is implemented. A method call to the ActiveX control requires approximately five times the number of instructions than a method call to the SocketWrench .NET managed class requires.

It is recommended that you use Visual Studio 2010 or later versions. Earlier versions of Visual Studio may require that you use those development tools with elevated privileges. If you are developing on Windows 10 using the SocketWrench .NET component, it is recommended that you use Visual Studio 2019 or later versions.

SocketWrench 12 includes assemblies that can target the .NET 4.0 Framework and later versions, including .NET 9.0.

Upgrading Projects

If you are upgrading from an earlier version of SocketWrench, you should find that most projects which have used SocketWrench 8.0 or later will require minimal code changes. If you have used hard-coded values instead of constant names for options, then you should review those values and update them appropriately. Some option values will have changed over the versions as new features have been added. If you are upgrading from a very early version of SocketWrench, you may find that there have been changes to method names and parameters which will need to be updated.

The SocketWrench .NET assembly modules have retained the same name; however, the platform specific 32-bit and 64-bit interop libraries have changed to **SocketTools12.Interop.dll**. This allows applications created using SocketWrench 12 to co-exist with applications created using earlier versions of SocketTools and SocketWrench.

If you are upgrading from SocketTools 8.0 and earlier versions, the path to the common assembly folder has changed. Earlier versions of SocketTools did not provide assemblies which target .NET 4.5 or later versions of the .NET Framework. The SocketWrench 12 assemblies are now installed in the folder

C:\Program Files (x86)\Common Files\SocketTools\12.0\Assemblies for each version of the .NET Framework which is supported.

If you are using the SocketWrench ActiveX control, your application should not attempt to reference the current version of the control and an earlier version within the same application. When upgrading to SocketWrench 12, first remove all references to the earlier version of the control, save the project and reload it. Then add the reference to the version 12 control, ensuring that the same object name is used. If you are creating an instance of the control dynamically by specifying its ProgID, such as using the **CreateObject** function, then it is recommended that you specify the version number as part of the ID. For example, to create an instance of the FTP control, use "SocketTools.FtpClient.12" and not simply "SocketTools.FtpClient". If the major version number is omitted, the latest version of the control will always be loaded.

The runtime license key has changed for SocketWrench 12, which may require you to redefine the value in your application when calling the control's **Initialize** method or function. As with previous versions of SocketWrench, you can use the License Manager utility to generate a file which contains the runtime key you should use. The version 10 and earlier runtime license keys are not valid for the version 12 components and an error will be returned if an invalid runtime key is specified.

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

It is possible to force SocketWrench to use earlier versions of TLS for backwards compatibility with older servers. With the ActiveX control and .NET class, this is done by explicitly setting the **SecureProtocol** property to specify the protocol version required. With the SocketWrench API, you would set the **dwProtocol** member of the SECURITYCREDENTIALS structure to specify the protocol version required. However, this is not generally recommended because using an older version of TLS (or any version of SSL) may cause servers to immediately reject the connection attempt.

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

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

ActiveX File Names

The file names of the ActiveX controls and their IDs have changed with the new version. The following table lists the new values which should be used in your application.

File Name	ProgID	Description
cswskx12.ocx	SocketTools.SocketWrench.12	Windows Sockets (SocketWrench) Control
cswsvx12.ocx	SocketTools.InternetServer.12	Internet Server Control

Library File Names

The file name for the SocketWrench library has changed with the new version. The following table lists the new name and import library which should be used.

File Name	Import Library	Description
cswskv12.ocx	cswskv12.lib	Windows Sockets (SocketWrench) Library

SocketWrench Product Evaluation

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

Redistribution Restrictions

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

Runtime Licensing

When you purchase a development license, a runtime license key will be generated for you which will be included in your applications. Normally this runtime key is managed automatically when the control is placed on a form or referenced in a project. However, there are situations in which the key must be explicitly passed to the control's Initialize method. In all cases, if the product is installed as an evaluation copy, the runtime license key will be defined as an empty string. If you have previously installed an evaluation copy of SocketWrench and then purchased a license, you can create the runtime license key using the License Manager utility.

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

SocketWrench Licensing and Redistribution

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

System Requirements

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

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

Version Information

The SocketWrench components and libraries have information embedded in them which provides version information to an installation utility. This information called the version resource, specifies the library or control's version number among other things. If you are using a third-party or in-house installation program, it is extremely important that the program knows how to use this information.

For example, if you are deploying an application which uses the SocketWrench DLL, the setup program must determine if that library has already been installed on the target system. If it has, it must compare the version resource information in the two libraries. It should only overwrite the library if the version that you have included with your application is later than the one installed on the system. An installation program which overwrites the library without checking the version number may cause other programs to fail unexpectedly on the end-user's system, which is obviously not desirable.

.NET Class Redistribution

For those applications created using SocketWrench .NET, the appropriate class libraries (assemblies) must be distributed along with the application. The following component files should be included with your application:

File Name	Description
SocketTools.SocketWrench.dll	The SocketWrench (Windows Sockets) component which provides a general purpose networking interface, allowing applications to connect to other systems and exchange data using the TCP/IP protocol. This assembly is added as a reference in the project.
SocketTools.InternetServer.dll	The Internet Server component which provides the framework for implementing an event-driven multithreaded server application. This assembly is added as a reference in the project.
SocketTools11.Interop.dll	The SocketTools runtime library. This library is shared by all of the SocketWrench and SocketTools class libraries. This library should

not be referenced directly in the project.

SocketTools11.TraceLog.dll	The SocketTools debugging library used to generate trace log files which record the low-level networking functions called and the data exchanged. This library only needs to be distributed if the debugging features of the class are used. This library should not be referenced directly in the project.

.NET Installation Directory

It is recommended that you install the SocketWrench .NET assemblies in the same directory as the application executable on the target system. It is not recommended that you install the assemblies in the Global Assembly Cache (GAC) or in the Windows system folder. The assemblies are built to target both the x86 and x64 platforms and can be used on either type of system. The SocketTools11.Interop.dll runtime library is platform specific and it is recommended that you install it in the Windows system folder. This is a shared library that is used by all of the SocketTools assemblies and should not be referenced directly in your project.

If you are deploying your application to a system running the 32-bit version of Windows, you should install the 32-bit version of the SocketTools11.Interop.dll library in the \Windows\System32 folder. If you are deploying your application to a system running the 64-bit version of Windows, you should install the 32-bit version of the runtime library in the \Windows\Syswow64 folder, and the 64-bit version of the library in the \Windows\System32 folder. If you are using the **Trace** related properties in your application, you should also include SocketTools11.TraceLog.dll library in your installation package and install it in the same folder as the runtime library.

The installer should always perform version checking to ensure that it is not overwriting a newer version of the runtime library with an older version. If your installer package creates a 32-bit executable and you're deploying a 64-bit application, the installer must be capable of detecting that it is running on a 64-bit system and can disable filesystem redirection to ensure that the 64-bit libraries are installed in the correct location. Consult the documentation for your installer to determine if it is 64-bit compatible.

ActiveX Control Redistribution

For those applications created using the SocketWrench ActiveX control, the **cswsx11.ocx** file must be distributed along with the application and the control must be registered by the installation program. The process of registration means that specific entries must be created in the system registry which provides information about the control such as the location of the OCX file. Fortunately, ActiveX controls are self-registering which means that the control has the ability to create or update those registry entries itself.

To automatically register a control when your application is being installed, the installation program must be capable of loading the control and calling specific functions which will update the registry. Most modern installation tools are capable of registering ActiveX controls. For custom installation programs, refer to the article [Installing and Registering ActiveX Controls](#) on the Microsoft Developers Network site.

It is possible to register ActiveX controls manually without the use of an installation program. This may be desirable in those situations where an application is being deployed internally or the developer does not want to create a setup program for a limited distribution. The tool used to manually register a control is named RegSvr32.exe and can be obtained from a number of places including the Visual Basic or Visual C++ CD-ROM. This utility accepts a command line argument which specifies the name of the control to register. For example:

```
C:>regsvr32 c:\windows\system32\cswsx11.ocx
```

A message box would be displayed indicating that the control was registered successfully. To prevent the message box from being displayed, use the /S option which tells the utility to function silently. If an error is

reported, typically the reason is that a required system DLL is missing or out of date.



COM registration requires account elevation under Windows Vista, Windows Server 2008 and later versions because it modifies the system registry. To register controls from the command prompt, you must run it with administrative privileges. From the Start menu, select All Programs > Accessories and right-click on the Command Prompt item. Select "Run as Administrator" from the context menu that is displayed.

If the ActiveX control is installed on a 64-bit version of Windows, the 32-bit control, which is used with Visual Basic 6.0 and other 32-bit development tools, is installed in the C:\Windows\Syswow64 folder and the 64-bit control is installed in the C:\Windows\System32 folder. When redistributing the ActiveX control, it is important to make sure that you are selecting the correct version, which is determined by the development tool used and the target platform. For example, if you are using Visual Basic 6.0, then you should only redistribute the 32-bit ActiveX control, regardless if the target system is the 32-bit or 64-bit version of Windows. This is because Visual Basic 6.0 can only create 32-bit programs and therefore can only reference 32-bit controls and libraries. When the application is installed on 64-bit Windows, it will be executed by the WoW64 subsystem which provides a 32-bit environment for the application.

ActiveX Installation Directory

The SocketWrench ActiveX control should typically be installed in the \Windows\System32 directory on the local machine. Some developers may prefer to install the control along with their application in a private directory. It is not recommended that developers take this approach unless COM redirection or registration-free activation is used because the full pathname of the control file is stored in the system registry when it's registered. If multiple applications install the same control in different directories, the actual control that will be used is the one that was last registered. This means that it is possible that an application will load an earlier version of the control than it was built with, which may result in unexpected or fatal errors.

COM redirection enables an application to isolate the controls that it uses, ensuring that the same version of the control which was used to build the application is loaded when the program is executed. To activate COM redirection, create an empty file named after the executable with a .local extension. For example, if the program is named MyProgram.exe then an empty file named MyProgram.exe.local should be created in the same directory as MyApp.exe. This binds the application to the local version of any controls which are installed in the same directory as the application. When an instance of the control is created, Windows will first search the application's directory, and then uses the standard search rules for locating the file. Note that COM redirection is not supported on Windows 95 or Windows 98.

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

Library Redistribution

For those applications created using the SocketWrench API, the **cswskv12.dll** file must be distributed along with the application. The library has no external dependencies, other than standard Windows libraries that are part of the base operating system. In particular, the SocketWrench library does not use the Microsoft Foundation Classes, nor does it require the Visual C++ Runtime library. The library is a standard Windows DLL and does not require COM registration.

If SocketWrench is installed on a 64-bit version of Windows, the 32-bit DLLs are installed in the C:\Windows\Syswow64 folder and the 64-bit DLLs are installed in the C:\Windows\System32 folder. It is important to make sure that you are selecting the correct version, which is determined by the development tool used and the target platform. For example, if you are using Visual Studio 2010 and target the Win32 platform, then you should only redistribute the 32-bit DLL, regardless if the target system is the 32-bit or 64-bit

version of Windows. This is because 32-bit programs can only reference 32-bit libraries. When the application is installed on 64-bit Windows, it will be executed by the WoW64 subsystem which provides a 32-bit environment for the application.

Library Installation Directory

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

About SocketTools 12

SocketWrench is just one of the components in SocketTools. In addition to the lower-level access that SocketWrench provides, SocketTools includes .NET assemblies, ActiveX controls and native Windows libraries for many of the popular Internet application protocols. There are several different editions of SocketTools available, and all editions provide royalty-free redistribution licensing and a thirty day money-back guarantee. Free evaluation copies can be downloaded from the Catalyst Development website at sockettools.com.

SocketTools .NET Edition

The SocketTools .NET Edition consists of managed code assemblies for use with .NET programming languages such as Visual Basic .NET, Visual C# and Delphi Prism. The product includes over twenty different classes which provide interfaces for various Internet protocols such as the File Transfer Protocol, Hypertext Transfer Protocol, Internet Message Access Protocol and Simple Mail Transfer Protocol. Using the .NET Edition you can easily transfer files, send and retrieve email messages, execute commands on servers and perform many other common tasks over the Internet. The SocketTools .NET classes are designed to be extremely simple to use without compromising performance, and are flexible enough to perform very complex tasks.

SocketTools ActiveX Edition

The SocketTools ActiveX Edition consists of ActiveX controls for use with visual development languages such as Visual Basic, Visual C++ and Delphi. It includes more than twenty controls that provide client interfaces for the major application protocols such as the File Transfer Protocol, Simple Mail Transfer Protocol, Domain Name Service and Telnet. Visual Basic 6.0 is fully supported and the components can be used with any development tool that supports COM and the ActiveX control specification. The network controls support both synchronous (blocking) and asynchronous modes of operation, as well as advanced trace debugging facilities. All of the controls can be used in multithreaded containers.

SocketTools Library Edition

The SocketTools Library Edition consists of standard dynamic link libraries, and can be used by virtually any Windows programming language that can call functions exported from a Windows DLL. It includes more than twenty libraries that provide client interfaces for application protocols such as the File Transfer Protocol, Simple Mail Transfer Protocol and Telnet protocol. The API for the Library Edition is implemented with a simple elegance that makes it easy to use with any language, and is not just for C++ programmers. All of the libraries are thread-safe and can be used in multithreaded applications.

Technical Support

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

Standard Support

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

[Frequently Asked Questions](#)

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

[Knowledge Base](#)

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

[Online Help](#)

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

Priority Support

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

SocketWrench License Agreement

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

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

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

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

1. LICENSE GRANT

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

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

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

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

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

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

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

2. EVALUATION USE

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

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

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

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

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

3. REDISTRIBUTION RIGHTS

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

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

You may:

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

You may not:

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

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

4. LICENSE TRANSFERS

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

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

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

5. OWNERSHIP

The Software is licensed, not sold.

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

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

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

6. THIRD-PARTY SERVICES

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

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

You acknowledge and agree that:

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

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

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

7. CATALYST SERVICES

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

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

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

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

8. WARRANTY DISCLAIMER

THE SOFTWARE IS PROVIDED "AS IS".

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

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

9. LIMITATION OF LIABILITY

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

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

10. EXPORT COMPLIANCE

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

11. GOVERNING LAW

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

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

12. GENERAL

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

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

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

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

SocketWrench Copyright

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

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

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

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

SocketWrench Developer's Guide

With the acceptance of TCP/IP as a standard platform-independent network protocol, and the explosive growth of the Internet, the Windows Sockets API (application program interface) has emerged as the standard for network programming in the Windows environment. This guide will introduce the basic concepts behind Windows Sockets programming and get you started with your first application created with SocketWrench. It is assumed that the reader is familiar with Visual Basic and has installed the SocketWrench control.

Designed for the professional software developer, SocketWrench is optimized for the Windows platform and implements secure protocols with support for up to 256-bit encryption. This new release includes both ActiveX controls, standard dynamic link libraries (DLLs) and C++ classes in the same package, along with new samples and over 400 pages of documentation. SocketWrench provides all of the features, documentation and technical support needed to develop complete Internet applications, without the complexities of learning the Windows Sockets API or working around the limitations of other Internet components.

SocketWrench is part of a larger product called [SocketTools](#) which includes a complete collection of controls and libraries for many of the popular Internet application protocols such as FTP, HTTP, POP3, IMAP4, SMTP and TELNET. Secure versions of these components are also available that support both standard and secure network connections using the standard TLS protocols. You'll find the same features, functionality and stability in the SocketTools package without having to learn how to implement complex application protocols or decipher cryptic standards documents. With SocketTools, adding features to transfer files, send or retrieve emails and access web pages can be done in just a few minutes. Instead of reinventing the wheel, you can spend your time working on your core application and increasing your productivity without sacrificing the features that your users expect.

To learn more about SocketTools family of products, please visit the Catalyst Development website at sockettools.com

Windows Sockets API

The Windows Sockets specification was created by a group of companies, including Microsoft, in an effort to standardize the TCP/IP suite of protocols under Windows. Prior to Windows Sockets, each vendor developed their own proprietary libraries, and although they all had similar functionality, the differences were significant enough to cause problems for the software developers that used them. The biggest limitation was that, upon choosing to develop against a specific vendor's library, the developer was "locked" into that particular implementation. A program written against one vendor's product would not work with another's. Windows Sockets was offered as a solution, leaving developers and their end-users free to choose any vendor's implementation with the assurance that the product will continue to work.

There are two general approaches that you can take when creating a program that uses Windows Sockets. One is to code directly against the API. The other is to use a component which provides a higher-level interface to the library by setting properties and responding to events. This can provide a more "natural" programming interface, and it allows you to avoid much of the error-prone drudgery commonly associated with sockets programming. By including the control in a project, setting some properties and responding to events, you can quickly and easily write an Internet-enabled application. And because of the nature of custom controls in general, the learning curve is low and experimentation is easy. SocketWrench provides a comprehensive interface to the Windows Sockets library and will be used to build a simple client-server application in the next section of this document. Before we get started with the control, however, we'll cover the basic terminology and concepts behind sockets programming in general.

Transmission Control Protocol

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

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

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

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

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

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

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

User Datagram Protocol

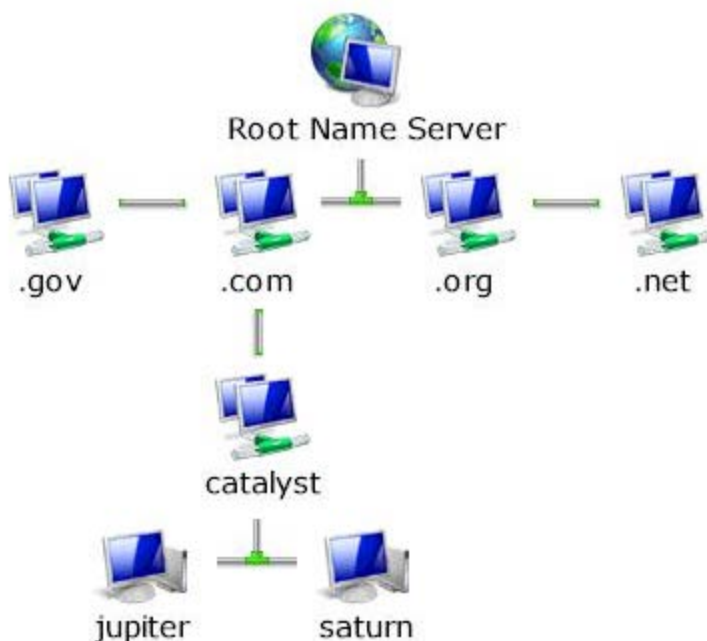
Unlike TCP, the User Datagram Protocol (UDP) does not present data as a stream of bytes, nor does it require that you establish a connection with another program in order to exchange information. Data is exchanged in discrete units called datagrams, which are similar to IP datagrams. In fact, the only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

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

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

Hostnames

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



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

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

Service Ports

In addition to the IP address of the remote system, an application also needs to know how to address the specific program that it wishes to communicate with. This is accomplished by specifying a service port, a 16-bit number that uniquely identifies an application running on the system. Instead of numbers, however, service names are usually used instead. Like hostnames, service names are usually matched to port numbers through a local file, commonly called services. This file lists the logical service name, followed by the port number and protocol used by the server.

A number of standard service names are used by Internet-based applications and these are referred to as well-known services. These services are defined by a standards document and include common application protocols such as FTP, POP3, SMTP and HTTP.

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

Sockets

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

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

Synchronous and Asynchronous Sockets

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

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

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

Synchronous Sockets

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

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

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

Asynchronous Sockets

SocketWrench facilitates the use of asynchronous sockets by generating events when appropriate. For example, an **OnRead** event occurs whenever the remote host writes on the socket, which tells your application that there is data waiting to be read. The use of non-blocking sockets will be demonstrated in the next section,

and is one of the key areas in which an ActiveX control has a distinct advantage over coding directly against the Windows Sockets API.

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

Best Practices

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

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

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

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

socket operations under some circumstances. However, switching between blocking and non-blocking mode can make the application more complex and difficult to debug. It is important to note that the warning regarding blocking sockets also applies here.

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

Client-Server Application Model

Programs written to use TCP are developed using the client-server model. As mentioned previously, when two programs wish to use TCP to exchange data, one of the programs must assume the role of the client, while the other must assume the role of the server. The client application initiates what is called an active open. It creates a socket and actively attempts to connect to a server program. On the other hand, the server application creates a socket and passively listens for incoming connections from clients, performing what is called a passive open. When the client initiates a connection, the server is notified that some process is attempting to connect with it. By accepting the connection, the server completes what is called a virtual circuit, a logical communications pathway between the two programs. It's important to note that the act of accepting a connection creates a new socket; the original socket remains unchanged so that it can continue to be used to listen for additional connections. When the server no longer wishes to listen for connections, it closes the original passive socket.

To review, there are five significant steps that a program which uses TCP must take to establish and complete a connection. The server side would follow these steps:

1. Create a socket.
2. Listen for incoming connections from clients.
3. Accept the client connection.
4. Send and receive information.
5. Close the socket when finished, terminating the conversation.

In the case of the client, these steps are followed:

1. Create a socket.
2. Specify the address and service port of the server program.
3. Establish the connection with the server.
4. Send and receive information.
5. Close the socket when finished, terminating the conversation.

Only steps two and three are different, depending on if it's a client or server application.

Secure Network Communication

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

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

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

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

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

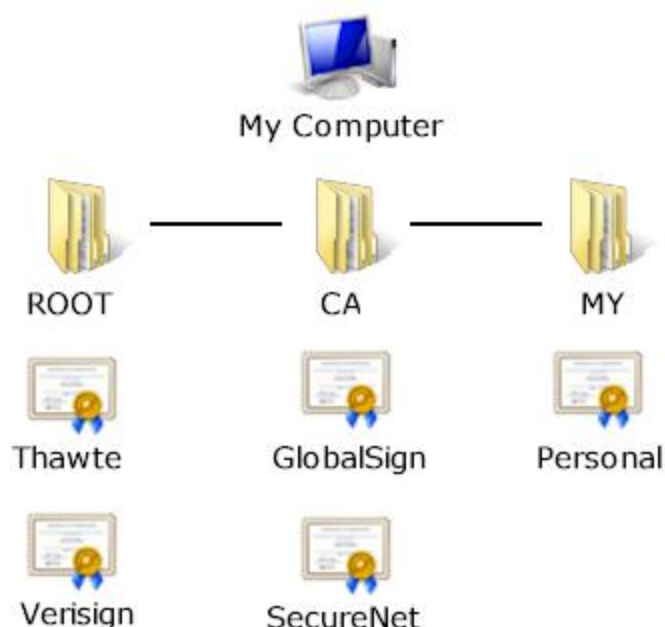
After the handshake phase has completed, the client may choose to examine the digital certificate that has been returned by the server. The information contained in the certificate includes the date that it was issued, the date that it expires, information about the organization who issued the certificate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

choose to terminate the connection and warn the user.

Digital Certificates

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

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



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

In general, the one situation where certificate management becomes important is when you want to develop

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

Debugging Applications

One of the issues that every developer has to contend with are problems that arise in an application after it's been distributed to end-users. And errors related to Windows Sockets programming can be even more difficult to track down because there are so many variables involved (such as the platform, operating system version, system configuration, and so on). To address these difficult problems, the SocketWrench control has the built-in ability to log the Windows Sockets API function calls that are made. There are three properties related to function tracing: **Trace**, **TraceFile** and **TraceFlags**. Setting these properties allows your application to dynamically manage function tracing features available to the control. The **Trace** property is a boolean flag which simply enables or disables the function tracing feature. The **TraceFile** property specifies the name of a trace log file in which each function and its parameters will be written. If this property is not explicitly set, then a file named CSTRACE.LOG will be created in the system's temporary directory (the directory specified by the TEMP environment variable). The **TraceFlags** property specifies what type of logging will be performed by the control, and may be set to one of four values: 0 (TRACE_INFO) in which all functions will be logged, 1 (TRACE_ERROR) in which only errors will be logged, 2 (TRACE_WARNING) in which case both warnings and errors will be written to the log file, and 4 (TRACE_HEXDUMP) in which all functions will be logged, together with ASCII and hexadecimal displays of all data that is sent or received on sockets. By default, all functions calls are logged by the control (TRACE_INFO).

For the SocketWrench library there are two functions related to function tracing: **InetEnableTrace** and **InetDisableTrace**. The arguments to **InetEnableTrace** are equivalent to the **TraceFile** and **TraceFlags** properties of the controls, as described above. Calling **InetEnableTrace** is equivalent to setting Trace = True, and calling **InetDisableTrace** is equivalent to setting Trace = False.

The trace file has the following format:

```
VB6 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process that is being traced (in this case, it is Visual Basic 6.0). The second column identifies if the trace record is reporting information, a warning, or an error. What follows is the name of the function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record (the value is placed inside brackets).

When reading a trace log, there are two common things that you will see:

1. The error code 10035, which corresponds to the Winsock error WSAEWOULDBLOCK is a normal occurrence on connect calls, and should not be taken as a cause for concern by itself.
2. The normal return value for a select call is greater than zero, typically a value of one. A select call that returns zero usually indicates a timeout.

If parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address) it is recorded as a hexadecimal value preceded with "0x". A special type of pointer, called a null pointer, is recorded as NULL. Those functions which expect socket addresses are displayed in the following format: aa.bb.cc.dd:nnnn

The first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. Note that in the second line of the above example, the control is attempting to connect to a system with the IP address 192.0.0.1 on port 1234.

There are several ways that you could incorporate function tracing in your software. The simplest would be a menu item or a command line switch (like /DEBUG) in which the **Trace** property would be set to True. A more complex approach would be to include a dialog or property sheet which allows the user to specify the log file name and tracing options. When an end-user calls for technical support and is encountering a problem that

you think may be network related, you can instruct them to enable the tracing feature and then email or fax you a copy of the log file. In turn, if it is a problem that you don't understand, you can send the log file to a support technician who can analyze the log and provide you with additional information about what may be going on inside your application.

Remember that if you do not use the tracing features at any time during the execution of your program, there is no additional performance penalty. If you do enable tracing at some point, the tracing library will be loaded and memory will be allocated by the logging functions. These functions open, append to the trace log, flush and then close the log file for each Windows Sockets function call that is made. This insures that the last function called is logged in case of a general protection fault or other abnormal termination of the program. However, because of the file I/O overhead, it's recommended that your program rename or remove the log file before beginning a new trace.

SocketTools Namespace

Classes

Class	Description
InternetDialer	The InternetDialer class provides a way for client applications to connect to the Internet using Microsoft Windows Remote Access Services (RAS). To use this class, the dial-up networking software must be installed on the local system. For access to the Internet, the TCP/IP protocol must be installed and configured. The class may configured to use either the SLIP or PPP protocols, depending on the requirements of the service provider. Refer to your system documentation for information about installing and configuring dial-up networking on your system.
InternetServer	The InternetServer class provides a simplified interface for creating event-driven, multithreaded server applications using the TCP/IP protocol. The interface is similar to the SocketTools.SocketWrench class, however it is designed specifically to make it easier to implement a server application without requiring the need to manage multiple socket classes. In addition, the class supports secure connections using TLS 1.2.
SocketWrench	The SocketWrench class is a general purpose networking class used to develop Internet and intranet applications using the TCP/IP protocol. With SocketWrench, you can create both client and server applications, as well as send and receive UDP datagrams. SocketWrench also supports secure connections using the Transport Layer Security (TLS) protocols. Enabling the security features of the class is done by setting a single property and does not require another .NET class to implement the encryption.

SocketWrench Class Overview

Before getting started with using SocketWrench, there are some general concepts used throughout the documentation which are helpful to understand. If you are new to network programming, you are also encouraged to review the [general concepts](#) in the Developer's Guide which covers these topics in more detail.

Connections and Sessions

One of the first general concepts that you'll encounter when developing Internet applications is that most programs act as either a client or a server. In simplest terms, a server is a program which is designed to perform specific functions on behalf of another program. A client is a program which is designed to request information from a server and then present that information to a user. It is common for one server to be able to interact with many clients, with each client functioning independently of one another. The interaction between a client and server can be broken down into several discrete steps:

- The client program attempts to connect to the server
- The server program accepts the connection
- The client sends a request to the server to perform some function
- The server processes the request, returning information to the client
- The client receives the information from the server and processes it
- The client disconnects from the server

When a client wants to request information from a server, the first step that it needs to take is to establish a connection. This is someone analogous to calling someone up on the telephone. You pick up the telephone, dial a number and wait for the other person to answer the phone and begin the conversation. With SocketWrench, the **Connect** method is what is used to begin the process of establishing the connection with the server. The host name or address tells the control what server it should be connecting to, just as the telephone number is used to specify who you want to talk to. The **Disconnect** method disconnects the program from the server, and is similar to saying goodbye and hanging up the telephone.

This complete process, from establishing the connection to disconnecting from the server, is typically referred to as a session. During a single session, the client may send one request, or it may choose to send several requests before terminating the connection.

Consider a web server such as the one that hosts the SocketTools website. That server is responsible for providing clients with the web pages and other content that they request. The client could be any browser, such as Microsoft Edge or Google Chrome. When you enter an address, such as <http://sockettools.com>, it instructs the browser to request the index page for the website from the server. The server retrieves the contents of that page and sends it back to the client as data. The client receives that data and displays it to the user. This is an example of a client/server session.

Host Names and Ports

Part of establishing a connection with a server is knowing the name of the server to connect to, and the port number for the service it is providing. Host names are strings which can be used to identify a server, similar to how a telephone number is used to specify who it is that you want to call. Everyone who has used a web browser is familiar with host names, such as sockettools.com or microsoft.com. In addition to host names, you can also use Internet addresses which are a series of four numbers separated by periods. For example, 192.168.0.10 would be an Internet address, also referred to as an IP address. The SocketWrench class has two properties, **HostName** and **HostAddress**, which can be used to specify the name or address of a server. You can also specify the host name or address as an optional argument to the **Connect** method, if you prefer.

In addition to a host name or address, a client program also needs to know what port number it should use to establish the connection. You can think of port numbers like the extension for a telephone number. Just as an

extension may be used to contact different employees using the same telephone number for a company, the port number may be used to connect to different services available on the same server. Port numbers are a way to distinguish between the different services available, and each protocol has a unique port number assigned to it. For example, a webserver uses port 80 to accept connections, while an FTP server uses port 21. The **RemotePort** property can be used to specify a port number, or the port number can be passed as an optional argument to the **Connect** method.

One important thing to keep in mind is that host names and URLs (Uniform Resource Locators) are not the same thing. For example, <http://sockettools.com> is not a valid host name. URLs include information about the protocol, the host name or address, the port number and the resource to access. When using the **Connect** method or setting the **HostName** property, make sure that you specify only the host name portion of the URL, such as sockettools.com.

Asynchronous Sessions

The SocketWrench .NET class has been designed to work in one of two basic modes of operation, establishing either a synchronous or asynchronous connection. The default mode of operation is synchronous, which is also referred to as a "blocking" connection. In this mode, it will wait for the requested operation to complete on the server or until the timeout period expires. For example, when the **Connect** method is called, SocketWrench will wait until the connection has completed before returning control to your program and the next statement is executed. The second mode, which is asynchronous or "non-blocking", causes the class to resume execution of your program immediately without waiting for the operation to complete. In that case, your program is notified through events that a particular operation has completed. For example, when the **Connect** method is called, it will immediately return and when the connection has completed, the **OnConnect** event will fire.

The class uses the **Blocking** property to determine if it should operate synchronously or asynchronously. In most cases, it is preferable to use the default mode, which is to establish a synchronous connection. Unless your application is written to specifically handle the various asynchronous network events, there can be unexpected results. For example, consider the following code:

```
Dim Socket As New SocketTools.SocketWrench
Dim strBuffer As String

If Socket.Connect("api.sockettools.com", 80) Then
    ' Send the GET command to the web server
    Socket.WriteLine("GET /test")

    ' Read the response from the server and store it in a
    ' a string buffer
    If Socket.ReadStream(strBuffer, True) Then
        TextBox1.Text = strBuffer
    End If

    ' Disconnect from the server
    Socket.Disconnect()
Else
    ' The connection attempt has failed, display an error
    MsgBox(Socket.LastErrorString, MsgBoxStyle.Exclamation)
    Exit Sub
End If
```

In this example, SocketWrench being used to establish a connection to the sockettools.com webserver. If the connection is successful, then the program sends a command to the server requesting a resource using the GET command and then reads the data returned by the server, displaying it in a TextBox control. If the connection attempt fails, a message box is displayed and the subroutine is exited. This code is fairly straightforward and would work as expected with a synchronous connection where the **Blocking** property is set to

true. However, if an asynchronous connection was used then it is very likely this code would fail unexpectedly. Why? Because the **Connect** method returns immediately in asynchronous mode, without waiting for the connection to actually complete. In this case, the code following the **Connect** method would need to be changed and moved out of that code block and into the various event handlers such as **OnConnect** and **OnRead**.

When SocketWrench is in blocking mode, the **Timeout** property is used to determine the amount of time that the control should wait for the operation to complete. The default timeout period is 20 seconds, however this can be set lower or higher as needed. To cancel a blocking operation and resume execution of the program, use the **Cancel** method.

In general, unless you have a specific need to use SocketWrench in asynchronous mode, we recommend that you use blocking connections. Asynchronous sessions are more complex to code for, have a greater tendency to introduce errors into the logical flow of a program and can be more difficult to debug. If you wish to perform multiple network operations at the same time, it is preferable to create multiple threads rather than attempt to manage multiple asynchronous sessions in a single thread. In addition, there is additional overhead imposed when using asynchronous sessions due to the event handling mechanism.

It should also be noted that certain high-level methods will always cause the class to block during execution, regardless of what mode it is in. An example of this is the **ReadStream** method, which reads an arbitrarily large stream of bytes from the remote host and stores it in a string or byte array buffer. When using SocketWrench in asynchronous mode, you need to make sure that you only use the lower-level methods such as **Read** and **Write**, and do not attempt to perform a blocking socket operation inside an asynchronous event handler.

Common Properties

Property Name	Description
AutoResolve	Determines if host names and IP addresses are automatically resolved.
Blocking	Gets and sets the blocking state of the class instance.
HostAddress	Gets and sets the IP address of the remote host.
HostName	Gets and sets the name of the remote host.
IsBlocked	Determine if the class instance is blocked performing an operation.
IsConnected	Determine if the class instance is connected to a remote host.
IsReadable	Determine if data can be read from the remote host without blocking.
IsWritable	Determine if data can be sent to the remote host without blocking.
LastError	Gets and sets the last error that occurred on the class instance.
LastErrorString	Return a description of the last error to occur.
LocalAddress	Return the IP address of the local host.
LocalName	Return the name of the local host.
RemotePort	Gets and sets the port number for a remote connection.
Secure	Set or return if a connection to the server is secure.
Status	Return the current status of the session.
ThrowError	Enable or disable error exceptions being generated by class methods.
Timeout	Gets and sets the amount of time until a blocking operation fails.
Trace	Enable or disable socket function level tracing.
TraceFile	Specify the socket function trace output file.
TraceFlags	Gets and sets the socket function tracing flags.
Version	Return the current version of the class library.

Many of the SocketWrench .NET class properties are similar to those used by the ActiveX control. Developers are familiar with previous versions of SocketWrench should find that this significantly reduces the amount of time required to port an existing application to the .NET framework. It is recommended that you also review the Technical Reference material for specific information about a particular property.

AutoResolve

The **AutoResolve** property controls how host names are resolved by the class whenever the `HostName` or `HostAddress` properties are set. By default, the property is set to **false**, which means that the class instance does not attempt to resolve the host name until a connection attempt is made. If the property is set to **true**, then the class will immediately attempt to resolve the host name into an IP address. Note that this can cause the class to block for several seconds and negatively affect the performance of your program. In most cases, this property should be set to **false**.

Blocking

The **Blocking** property determines whether or not the class operates in blocking (synchronous) mode or non-

blocking (asynchronous) mode. In blocking mode, the class waits for a given operation to complete before returning control to your application and executing the next statement. In non-blocking mode, control is immediately returned to the program without waiting for the operation to complete. In this case, events are used to notify the application that a specific operation has completed.

In general, using the class in blocking mode means that your code is going to be structured in a top-down fashion. For example, when establishing a connection with a remote system, your program will block until the connection has completed or has timed out. In non-blocking mode, your code is event driven and must implement event handlers to process those event notifications.

It is recommended that you only establish a non-blocking connection when you understand the implications of doing so and it is required by your application. If you require multiple instances of the class to establish connections to different servers, it is preferable to create a multithreaded application rather than attempt to use multiple instances of the class in a single thread.

HostAddress

The **HostAddress** property is used to specify the IP address of a remote host to connect to. The address should be given in dot notation, which is four numbers separated by periods (e.g.: 192.168.0.10). If the **AutoResolve** property is set to **true**, setting this property will force the class to immediately resolve the address into a host name. Note that if you attempt to set this property to the value of a host name, an exception will be thrown indicating that the property value is invalid.

HostName

The **HostName** property is used to specify the name of a remote host to connect to. This property will accept either host names or IP addresses. If an IP address is specified, then setting this property is similar to setting the **HostAddress** property. If the **AutoResolve** property is set to **true**, setting this property will force the class to immediately resolve the host name into an IP address. The value of this property is used as the default host name when the **Connect** method is called.

IsBlocked

The **IsBlocked** property returns **true** if the class instance is currently performing a blocking operation. This can be used in conjunction with the **Status** property to determine if the class can be used to issue a command to the server or perform some other operation. When the **IsBlocked** property returns **false** and the **Status** property returns a value of zero or one, the class instance is in either an inactive or idle state. If the program attempts to perform another operation while a blocking operation is in progress, the error **errorOperationInProgress** is returned.

IsConnected

The **IsConnected** property returns **true** if a connection has been made with a server, otherwise it will return **false**. The property is read-only, and any attempt to set it to a value will result in an error. To establish a connection, refer to the **Connect** method.

IsReadable

The **IsReadable** property returns **true** if there is data available to read using the **Read** method. If the property returns **false**, then there is no data available to be read. In this case, if the **Blocking** property is set to **true**, calling the **Read** method will cause the class to block until data arrives or the timeout period is exceeded; otherwise, it will fail and return the error **stErrorOperationWouldBlock**. Note that this property can only be used to determine if there is data available to be read, not the amount of data.

IsWritable

The **IsWritable** property returns **true** if the class can successfully write data using the **Write** method. If the property returns **false**, then the socket's internal buffers are full and cannot accept any more data until the

remote host reads some of the data that has already been written. In this case, if the **Blocking** property is set to **true**, the **Write** method will cause the class to block until the data can be written or the timeout period is exceeded; otherwise, it will fail and return the error `stErrorOperationWouldBlock`. Note that this property can only be used to determine if some data can be written, not the amount of data.

LastError

The **LastError** property returns a numeric value which identifies the last error that occurred. This property may be set to zero, which will clear the last error code. Note that setting this property to a non-zero value will have the effect of raising that error, which must be handled by the application. Refer to the Technical Reference for a complete list of error codes and their description.

LastErrorString

The **LastErrorString** property returns a description of the last error that occurred, and corresponds to the value of the **LastError** property. This property is typically used by an application to display a message box to the user or include information about the error in a log file. Note that the error description will be in English, regardless of the current locale settings.

LocalAddress

The **LocalAddress** property returns the IP address of the local host. Note that if the system is behind a router which uses Network Address Translation (NAT) then the IP address returned will be the address of the system on the local network, not the external WAN address assigned to the router.

LocalName

The **LocalName** property returns the fully qualified domain name of the local host, if that information is available. If the class is unable to determine the domain name for the local system, then it will return the machine name as it was configured in the Windows operating system.

RemotePort

The **RemotePort** property is used to specify the port number used to establish a connection with the remote host. Valid port numbers are in the range of 1 through 65535, and assigning the property a value greater than this will result in an error. This property value is used as the default port number when the **Connect** method is called.

Secure

The **Secure** property determines if the class should establish a secure connection to the server. The default value for this property is **false**, which specifies that a standard connection should be established. If this property is set to **true**, then the class will attempt to establish a secure connection using the TLS 1.2 or 1.3 protocol.

Status

The **Status** property returns a numeric value which identifies the current state of the class instance. A value of zero indicates that no connection has been established with the class. A value of one indicates that the class is in an "idle" state, waiting to process the next request or send a command to the server. Values greater than one indicates that instance of the class is actively performing some operation. Refer to the Technical Reference documentation to determine what each state value means.

ThrowError

The **ThrowError** property is used to determine how errors are reported by the class when calling a method. The default value is **false**, which specifies that errors should be returned as values from the method call and the class instance should not throw an exception. If this property is set to **true**, then methods will throw an exception whenever an error is encountered. This can be useful if you want to implement an exception

handler for any error conditions rather than checking the return value from each method call.

Timeout

The **Timeout** property used to determine how long the class instance will wait for a blocking operation to complete before returning control to the application. The default value for the property in most cases is 20 seconds. The **Timeout** property is only used when the **Blocking** property is set to **true**.

Trace

The **Trace** property is used to enable or disable the trace logging features of the class. When the property is set to **true**, that instance of the class will record all of the networking function calls that it makes, and depending on the trace level, the data exchanged between the client and server. To enable trace logging, you must include the trace library **SocketTools11.TraceLog.dll** with your application. If this library cannot be loaded, the value of the **Trace** property value will be ignored.

TraceFile

The **TraceFile** property is used to specify the name of a file that will contain the trace logging data generated when the **Trace** property is set to **true**. This property should be set prior to setting the **Trace** property.

TraceFlags

The **TraceFlags** property is used to specify the amount of information that is recorded by the trace logging facility. The default value **traceDefault** specifies that all of the networking function calls should be logged, along with their arguments and return values (this is the same as specifying the **traceInfo** option). The following values are used:

Trace Option	Description
traceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
traceError	Only those function calls which fail are recorded in the trace file. Functions which are successful or only return values which indicate a warning are not logged.
traceWarning	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file. Successful function calls are not logged.
traceHexDump	All functions calls are written to the trace file, plus all the data that is sent or received is displayed in both ASCII and hexadecimal format. This is useful for examining the actual byte stream that is exchanged between the application and the remote host.

Version

The **Version** property returns the current version of the class instance as a string. This can be used by the application to check that the correct version of the class has been installed on the local system.

Common Methods

Method Name	Description
Accept	Accept a client connection. This is used by server applications.
Cancel	Cancels the current blocking socket operation.
Connect	Establish a connection with a remote host This is used by client applications.
Disconnect	Terminate the connection with the remote host.
Initialize	Initialize the class and load the networking interface.
Listen	Listen for client connections. This is used by server applications.
Read	Read a block of data from the remote host and return it in a string or byte array.
ReadLine	Read a line of text from the remote host and return it in a string.
Reset	Reset the internal state of the class and terminate any active connections.
Uninitialize	Uninitialize the class and release resources allocated for the class instance.
Write	Send a block of data to the remote host.
WriteLine	Send a line of text to the remote host, terminated with a newline.

Most methods will return a value of **true** if they are successful, or a value of **false** if the method fails for some reason. The error code which specifies the cause of the failure is returned by the **LastError** property. A human readable description of the error can be obtained by getting the value of the **LastErrorString** property. There are some exceptions to this rule, such as the **Read** and **Write** methods which return the number of bytes read or written, and a value of -1 if there was an error. These exceptions are noted in the Technical Reference section.

Many of the SocketWrench .NET class methods are similar to those used by the ActiveX control, however the return values have changed in some cases. It is recommended that you review the Technical Reference material for specific information about a particular method.

Note that the way that a method indicates an error condition is affected by the **ThrowError** property. This allows errors to be handled in one of two ways, depending on the personal preferences of the developer and requirements of the application. If the property is set to **true**, then methods will throw an exception when an error occurs and it is the responsibility of the application to implement an exception handler. If the property is set to **false**, then the method will not throw an exception and will simply return a value indicating success or failure. The default value for the property is **false**.

Accept

The **Accept** method is used by server applications to accept an incoming client connection. After the connection has been accepted, the server may begin to send and receive data on the socket. Typically a new instance of the SocketWrench class is created for each client connection and the handle of the listening server socket is passed as an argument to the **Accept** method. This allows the server to continue to listen for new connection requests, and the new instance of the class becomes responsible for exchanging data with the client.

Cancel

The **Cancel** method cancels the current blocking operation being performed by the class. For example, if the **Connect** method has been called, the **Cancel** method will cancel the connection attempt. When this happens, the **OnCancel** event will fire and the blocking method will return with the error **errorOperationCanceled**.

Once an operation has been canceled, it is important to allow the application to unwind the stack and resume execution at the point where the blocking method returns. For example, you should not call the **Cancel** method and then perform another blocking operation in any event handler until after the blocking method returns.

Connect

The **Connect** method is used to establish a connection with a remote host, and is typically one of the first methods called by the program. There are several overloaded implementations of this method. If no arguments are specified, then the method will use the values of the **HostName** or **HostAddress** and **RemotePort** properties as the default. If the **Blocking** property is set to **true**, then the method will return after the connection has been established, or after the timeout period has been exceeded. If the **Blocking** property is set to **false**, the method will return immediately and the application must wait for either the **OnConnect** or **OnError** event to fire before performing any other operation on the socket.

Disconnect

The **Disconnect** method terminates the current connection and releases some of the resources allocated by the class for the network connection. For every call to the **Connect** method, there should be a matching call made to the **Disconnect** method when the connection is no longer needed.

Initialize

The **Initialize** method explicitly initializes the class, loading the appropriate networking interface, validating the runtime license key and performing other internal initialization functions. Your program must call the **Initialize** method before setting any properties or calling any other method. For each call to the **Initialize** method, there should be a matching call made to the **Uninitialize** method when that instance of the class is no longer being used. If the **Initialize** method is called without specifying a valid runtime key, then the program will only execute on a system that has a valid development license. To redistribute your application, you must purchase a license and provide a valid runtime key. For more information, refer to the [Class Initialization](#) section of the Developer's Guide.

Listen

The **Listen** method is used by server applications to create a socket that will "listen" for incoming client connections. The server would then call the **Accept** method to accept the connection and begin communicating with the client. If the **Blocking** property is set to **false**, the **OnAccept** event will occur whenever a client attempts to connect with the server.

Read

The **Read** method is used to read data returned by the server in response to a command sent by the client. The type of data returned depends on the protocol being used. The first argument passed to the method should be a string or byte array which will contain the data that has been read when the method returns. The second argument should be an integer which specifies the amount of data to read, in bytes.

The **Read** method is different from most other methods in two important ways. Instead of returning true or false, the method returns the number of bytes read. If an error occurs, then the method will return -1. To determine the cause of the error, check the value of the **LastError** property. If there is no more data to be read and the server has closed its connection to your program, then the method will return 0.

In addition, the variable which will contain the data must be passed by reference to the method. In languages like Visual Basic.NET, this is automatically handled for you. However, other languages may require you to use a special syntax to indicate that the variable should be passed by reference rather than by value. For example, C# requires the use of the **ref** keyword to pass a variable by reference.

ReadLine

The **ReadLine** method is used to read a line of text from the socket, up to a terminating carriage return and linefeed sequence. This is similar to reading a line of text from a file, and the method will return **false** when there is no more data to read or an error has occurred. It is important to keep in mind that this method should only be used with textual data, and it can force the thread to block (even if the **Blocking** property is set to **false**) while it buffers data up to the end of line. It is recommended that this method only be used with synchronous socket connections.

Reset

The **Reset** method will reset the internal state of the class instance to its defaults, terminating any connection to a remote host and releasing resources allocated for the session. This method should only be used when the program needs to effectively abort any active connections and return to a known state. In most cases, it is preferable for the application to use the **Disconnect** method to cleanly terminate the session.

Uninitialize

The **Uninitialize** method is used to unload the networking interface and release those resources which have been allocated by the class. In most cases, it is not necessary to explicitly uninitialize the class because this is handled automatically by the class destructor.

Write

The **Write** method is used to send data to the server. The first argument passed to the method should be a string or byte array which will contain the data to be written. The second argument should be an integer which specifies the number of bytes of data in the string or byte array. The **Write** method is different from most other methods because instead of returning true or false, the method returns the number of bytes written. If an error occurs, then the method will return -1. To determine the cause of the error, check the value of the **LastError** property.

WriteLine

The **WriteLine** method is used to send a line of text to the server, terminated with a carriage return and linefeed. This is similar to writing a line of text to a file, and the method will return false if no data can be written to the socket. As with the **ReadLine** method, this method should only be used with textual data and it can force the thread to block while the data is being written (even if the **Blocking** property is set to **false**). It is recommended that this method only be used with synchronous socket connections.

Common Events

Event Name	Description
OnAccept	This event occurs when a remote host connects with the server.
OnCancel	This event occurs when a blocking operation is canceled.
OnConnect	This event occurs when a client connection is established.
OnDisconnect	This event occurs when a connection is terminated.
OnError	This event occurs when a control error occurs.
OnProgress	This event occurs during data transfer.
OnRead	This event occurs when data is available to be read.
OnTimeout	This event occurs when a blocking operation times out.
OnWrite	This event occurs when data can be written to the server.

The events generated by SocketWrench .NET can be divided into two general categories, asynchronous network events and status notification events. Events such as **OnConnect** and **OnRead** are examples of network events which are generated when the class instance is placed in non-blocking mode. Events such as **OnError** and **OnProgress** are examples of notification events which are designed to provide additional status information to your application.

All event arguments are packaged in a class derived from **EventArgs**, and passed to the caller along with an argument that specifies the instance of the class that generated the event. For more information about how to implement an event handler, refer to the section on [Event Handling](#) in the Developer's Guide.

When developing your event handlers, it is important to remember that the underlying event mechanism uses Windows messages and requires that the application process those messages. That means that events may not fire correctly if the application is executing code in a tight loop and no messages are being dispatched. Another consideration is that some functions can interfere with the normal operation of events. For example, the MsgBox function in Visual Basic will force event handling to be suspended until the user closes the message box. Single stepping through code in the debugger can also prevent events from being processed normally. To debug code in an event handler, it is recommend that you use methods such as writing diagnostic messages to the immediate (debugging) window or a log file, rather than more intrusive measures such as displaying a message box.

OnAccept

The **OnAccept** event is generated whenever a remote host (client) attempts to connect with your server application. The **Blocking** property must be set to false and the **Listen** method must be called to enable the socket to listen for client connections. An instance of the **AcceptEventArgs** class is passed to the event handler. The class has a public property named **Handle** which specifies the handle of the socket which is listening for connections. This value is passed to the **Accept** method in order to accept the connection, allowing the server to begin communicating with the client.

OnCancel

The **OnCancel** event is generated whenever a blocking operation has been canceled using the **Cancel** method. When this event fires, the class is about to return control to your application, and the blocked method will return with the error **errorOperationCanceled**. It is important to note that you should not perform another blocking operation while inside the event handler. Instead, allow the stack to unwind and return control to the calling method.

OnConnect

The **OnConnect** event is a networking event that indicates that the connection request has completed and the client has successfully established a connection with the remote host. This event is only generated when the **Blocking** property is set to **false**. If the class is used to establish a non-blocking connection, the application must wait for this event to fire before attempting to perform any other functions.

OnDisconnect

The **OnDisconnect** event is a networking event that indicates that the remote host has closed its connection to the client. When this event occurs, your program should attempt to read any remaining data and then call the **Disconnect** method to close its connection to the server. This event is only generated when the **Blocking** property is set to **false**.

It is important to note that a failure to check for any remaining data in the socket receive buffers in the **OnDisconnect** event handler can result in unexpected data loss. It is recommended that you call the **Read** method until it returns a value of zero, which indicates that there is no more data available to be read. The **IsReadable** property can also be used to determine if there is any data available to be read.

OnError

The **OnError** event occurs whenever an error is reported by the class. An instance of the **ErrorEventArgs** class is passed to the event handler. The class has two public properties, **Error** and **Description**, which return the numeric error code and a human readable description of the error. These values correspond to the **LastError** and **LastErrorString** properties. This event is typically used by applications to record any errors that occur, either as information for the user or for debugging purposes.

OnProgress

The **OnProgress** event occurs during blocking operations when the **ReadStream**, **StoreStream** and **WriteStream** methods are called, providing information to the application about the status of the data transfer. This event is typically used to update the user interface, such as setting the value of a progress bar control.

OnRead

The **OnRead** event is a networking event which occurs whenever there is data available to be read from the socket. This event is only generated when the **Blocking** property is set to **false**. An important consideration when handling the **OnRead** event is that this event is level-triggered. This means that the event will only fire once, and will not fire again even if more data arrives, until at least one byte of data has been read by the application. This is by design, to prevent the application from being flooded with event messages.

OnTimeout

The **OnTimeout** event is generated whenever a blocking operation has exceeded the amount of time specified by the **Timeout** property. When this event fires, the class is about to return control to your application, and the blocked method will return with the error **errorOperationTimeout**. It is important to note that you should not perform another blocking operation while inside the event handler. Instead, allow the stack to unwind and return control to the calling method.

OnWrite

The **OnWrite** event is a networking event which occurs whenever the remote host is ready to receive more data from your application. This event is only generated under one of two circumstances, and only when the **Blocking** property is set to **false**. The first is when a connection has initially been established, the **OnWrite** event will occur immediately after the **OnConnect** event. The second case is when a previous call to the **Write** method failed with the error **errorOperationWouldBlock**, indicating that no more data could be written to the socket at that time because the send buffer was full.

Class Initialization

When you begin developing your application using SocketWrench .NET, the first thing that must happen is the class instance must be initialized. The initialization method serves two purposes. It loads the networking libraries required to establish a connection and it validates the runtime license key that you provide. The runtime license key is a string of characters which identifies your license to use and redistribute SocketWrench. It is unique to your product serial number and must be used when redistributing your application to an end-user.

Creating an instance of the class with your runtime license key can be accomplished in one of two ways, depending on personal preferences and the design of your application. The simplest approach, and one familiar to developers who have used the SocketWrench ActiveX control, is to explicitly call the [Initialize](#) method in your code. The other more advanced approach is to define the [RuntimeLicense](#) attribute for the assembly that is referencing the class. This attribute is set in the AssemblyInfo.vb or AssemblyInfo.cs module that is part of the project. For more information on specific usage, refer to the Technical Reference.

Developers who are evaluating SocketWrench will not have a runtime license key and must pass an empty string to the **Initialize** method. This will enable that instance of the class to load on the development system during the evaluation period, but will prevent component from being redistributed to an end-user until a license has been purchased.

If you install the product with a serial number, the runtime license key will be automatically created for you during the installation process. If you have installed an evaluation copy of SocketWrench and then purchased a license, the license key can be created using the License Manager utility that was included with the product. Simply select the License | Header File menu option and select the programming language that you are using.

The runtime license key is normally stored in the Include folder where you installed SocketWrench and is defined in a file named SocketWrenchLicense which can be included with your application. For example, C# programmers would use the SocketWrenchLicense.cs header file while Visual Basic programmers would use the SocketWrenchLicense.vb module. The Visual Basic module would define the runtime license key as:

```
'  
' SocketWrench 12.0  
' Copyright 2026 Catalyst Development Corporation  
' All rights reserved  
'  
' This file is licensed to you pursuant to the terms of the  
' product license agreement included with the original software  
' and is protected by copyright law and international treaties.  
'  
  
Public Const SocketWrenchLicenseKey As String = ""
```

This could either be included with your Visual Basic.NET application or you could simply copy the string into your application. The class could then be initialized like this:

```
'  
' Initialize the control using the specified runtime license key;  
' if the key is not specified, the development license will be used  
'  
  
If Socket.Initialize(SocketWrenchLicenseKey) = False Then  
    MsgBox("Unable to initialize SocketWrench component")  
End If
```

A return value of **true** indicates that the class was initialized successfully. A return value of **false** indicates that the class could not be initialized with the specified runtime license key. The **LastError** property will contain the error code that indicates the exact cause of the error.

An application is only required to call the Initialize method once, but it must be called for each instance of the class that is created. It is safe to call the initialization method more than once, but for each time that it is called, you must call the Uninitialize method for that class before your program terminates. In other words, if you called Initialize at the beginning of your program, you must call Uninitialize before your program ends. The Uninitialize method performs any necessary housekeeping operations, such as returning memory allocated for the class back to the operating system. If there are any open connections at the time that the Uninitialize method is called, they will be aborted. After the class has been uninitialized, you must call the Initialize method again before setting any properties or calling any methods in that instance of the class.

SocketWrench .NET Overview

The SocketWrench .NET component can be used to perform a variety of Internet related programming tasks. Although the number of properties, methods and events may appear daunting, once you begin using SocketWrench in your own applications you'll find that the various methods and events are designed to work together in a cohesive fashion. If you are already familiar with using the SocketWrench ActiveX component, then you'll find the SocketWrench .NET interface to be very similar.

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

Protocols

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

Connections

The process of establishing a connection enables one program to communicate with another. Connection requests are made by client applications, and accepted by server applications. When the server accepts the connection request, the connection is completed. When you use the Connect method to successfully establish a connection to a server, a client session is created. SocketWrench uses a one-to-one relationship between an instance of a class and a client session. By creating multiple instances of the class, an application can create multiple client/server sessions if necessary. When you use the Listen and Accept methods to accept incoming client connections, a server session is created. As with client-side connections, each server session is handled by a single instance of the class which is typically created dynamically as remote clients connect to the server.

Sessions

A session refers to an active connection between a client and server program. This term is typically used interchangeably with connection; however in some cases a single session may involve multiple network connections, depending on the application protocol being implemented. When the session is no longer needed, the Disconnect method will terminate the connection to the remote host and release the resources allocated for that session. After that point, the session is no longer valid and subsequent function calls using the class cannot be made until another connection is established.

Authentication

Many servers require that clients authenticate themselves by providing user names and passwords. Different application protocols implement several different types of authentication, and some protocols may support more than one authentication method. It is the responsibility of the application to implement whatever authentication protocol is required by the remote host. If a secure connection is being established with a server, it also may require that a digital certificate (called a client certificate) be provided as part negotiating the secure session. SocketWrench supports the ability to specify client certificates by setting the **CertificateName** and **CertificateStore** properties.

Events

Developers who use programming languages such as Visual Basic will find the concept of events and event handling to be very familiar. In general terms, the SocketWrench documentation uses "event" to refer to a mechanism where the control notifies the application that an operation has completed, some action has taken place or a change in status has occurred. One example of an event is the **OnConnect** event, which is

generated whenever an asynchronous network connection is completed by the client. Another example is the **OnProgress** event, which is generated periodically to inform the application of its progress as it sends or receives data. To determine what events are available based on usage, refer to the technical reference documentation..

Class Overview

The SocketWrench control provides a simplified interface to the Windows Sockets API. It was designed to be easier to use, and to provide properties and methods which eliminate much of the redundant coding common to Windows Sockets programming. Developers who are working in languages other than C or C++ will find SocketWrench to be particularly useful. SocketWrench also supports creating client and server applications which use the TLS security protocol without any dependencies on third-party security libraries.

The following properties, methods and events are available for use by your application:

Initialize

Initialize the control and validate the runtime license key for the current process. This method is normally not used if the control is placed on a form in languages such as Visual Basic. However, if the control is being created dynamically using a function similar to `CreateObject`, then the application must call this method to initialize the component before setting any properties or calling any other methods in the control.

Connect

Connect to the remote host, using either a host name or IP address. When an application calls this method, it will be acting as a client. This method creates the socket and must be called before your application attempts to exchange data with a server. For an asynchronous session, set the `Blocking` property to `False`.

Listen

Begin listening for incoming client connections. When an application calls this method, it will be acting as a server. Once the `Listen` method returns, the socket is created and that socket handle is used by the `Accept` method to accept an incoming client connection. For an asynchronous session, set the `Blocking` property to `False`.

Accept

Accept a connection from a client. This method should only be called if the application has previously called the `Listen` method. If there is no client waiting to connect at the time this method is called, it will block until a client connects or the timeout period is reached.

Uninitialize

Unload the Windows Sockets library and release any resources that have been allocated for the current process. This is the last method call that the application should make prior to terminating. This is only necessary if the application has previously called the `Initialize` method.

Input and Output

When a TCP connection is established, data is sent and received as a stream of bytes. The following methods can be used to send and receive data over the socket:

Read

A low-level method used to read data from the socket and copy it to the string buffer or byte array provided by the caller. If the remote host closes the connection, this method will return zero after all the data has been read. If the method is successful, it will return the actual number of bytes written.

ReadLine

Read a line of text from the socket, up to an end-of-line character sequence or when the remote host closes the connection.

ReadStream

A high-level method used to read a stream of bytes and copy it to a string buffer or byte array provided by the caller. This method can be used to read an arbitrarily large amount of data in a single call.

Write

A low-level method used to write data to the socket. If the method succeeds, the return value is the number of bytes actually written.

WriteLine

Write a line of text to the socket, terminating it with an end-of-line character sequence.

WriteStream

A high-level method used to write a stream of bytes to the socket. This method can be used to write an arbitrarily large amount of data to the socket in a single call.

IsReadable

This property is used to determine if there is data available to be read from the socket.

IsWritable

This property is used to determine if data can be written to the socket. In most cases this will return True, unless the internal socket buffers are full.

Host Name Resolution

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

HostAddress

This property can be used to set the IP address for a remote system that you wish to communicate with. If the address is valid and matches an entry in the host table, the HostName property will be changed to match the address.

HostName

This property should be set to the name of the remote system that you wish to communicate with. If the name is found in the host table, the HostAddress property is updated to reflect the IP address of the host. Note that it is legal to assign an IP address to this property, but it is not legal to assign a host name to the HostAddress property.

Local Host Information

Several methods are provided to return information about the local host, including its fully qualified domain name, local IP address and the physical MAC address of the primary network adapter.

LocalName

Return the fully qualified domain name of the local host, if it has been configured. If the system has not been configured with a domain name, then the machine name is returned instead.

LocalAddress

Return the IP address of the local host. If a connection has been established, then the IP address of the network adapter that was used to establish the connection will be returned. This can be particularly useful for multihomed systems that have more than one adapter and the application needs to know which adapter is being used for the connection.

ExternalAddress

Return the IP address assigned to the router that connects the local host to the Internet. This is typically used by an application executing on a system in a local network that uses a router which performs Network Address Translation (NAT).

PhysicalAddress

Return the physical MAC address for the primary network adapter on the local system.

AdapterAddress

This array returns the IP addresses that are associated with the local network or remote dial-up network adapters configured on the system. The **AdapterCount** property can be used to determine the number of adapters that are available.

Application Design

SocketWrench .NET is designed to be flexible enough to address the needs of developers who have very basic needs, as well as those who have more complex requirements. As a result, the properties and methods for the class interface can be broken down into two general categories: a high level interface to perform common tasks, and a lower level interface which provides more control at the expense of being somewhat more complicated and requiring more coding. For example, consider the high level methods such as **ReadLine** and **WriteLine**. Using these methods, your application read and write lines of text to a socket much the same way that you would to a text file. You don't need to understand how the data is being exchanged, buffered and processed. Another example would be the **StoreStream** method which allows you to easily read a stream of data from the socket and store it in a file. The high level methods allow you to program against the component as though it is a "black box", where you can provide the input and process the output without concerning yourself with the details of what's going on behind the scenes.

However, in some cases it's necessary for an application to have more direct control over how the control operates or to take advantage of features that aren't explicitly supported by one of the higher level methods. As an example, there's the lower-level **Read** and **Write** methods which allow you to directly read and write blocks of data on the socket and manage the buffering and data processing directly in your application.

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

One of the common pitfalls that developers can encounter with SocketWrench is the inclination to over-design the application from the start, and then become frustrated because they don't yet have a clear picture of how all the pieces fit together. Start out with a basic design and then as you become more familiar with how the class works, expand on it.

Program Structure

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

- Initializing an instance of the class
- Connecting to the server
- Authenticating the client
- Performing one or more operations
- Disconnecting from the server
- Uninitializing the class instance

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

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

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

Secure Connections

SocketWrench .NET supports the ability to create secure connections using the Transport Layer Security (TLS) protocols. In most cases, it is as simple as setting the **Secure** property to **true** or specifying an additional option when the **Connect** method is called. In some cases, certain Internet application protocols have additional requirements in terms of how the secure connection is established. Secure connections may either be implicit or explicit, depending on the protocol. An implicit secure connection is one where the client and server begin negotiating the security options as soon as the connection is established. In most cases, a server which accepts secure implicit connections listens on a port number that is different from the default port it uses for standard, non-secure connections. An example of this is the Hypertext Transfer Protocol (HTTP) which accepts standard connections on port 80 and secure connections on port 443. When a client connects to port 443, the server automatically assumes that the client wants a secure connection.

On the other hand, an explicit connection requires that the client explicitly specify to the server that it wants a secure connection. Typically this is done by the client sending a command to the server that causes the server to begin negotiating with the client to establish a secure session. An example of this is the File Transfer Protocol (FTP), where the client can use the AUTH command to tell the server that it wants a secure connection. Servers may also support both explicit and implicit secure connections, based on which port the client connects to. SocketWrench supports both implicit and explicit secure connections. If the **Secure** property is set to **true** prior to calling the **Connect** method, then an implicit secure connection is established. Setting the **Secure** property to **true** after a connection has been established will cause SocketWrench to begin negotiating a secure connection at that time.

In addition to establishing a secure connection, you may also be required to provide additional authentication information to the server in form a client certificate. For example, a server may require that the client provide a certificate in addition to or instead of a username and password. To support this, your application must specify the security credentials for the client prior to establishing a connection. For more information, refer to the **CertificateStore** and **CertificateName** properties in the Technical Reference.

Sending and Receiving Data

SocketWrench .NET provides several methods for exchanging data between your application and the remote host. At the lowest level, this is done by calling the **Write** method for sending data and the **Read** method for receiving data. In most cases, these methods exchange data as a stream of bytes without any regard for the actual content. It is important to note that if the data being read or written is binary, it is recommended that applications use byte arrays, not strings, to store the data in.

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

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

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

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

- Exchange the data as fixed length structures. This is the simplest approach, and has very little or no overhead. The client and server can either use predefined values, or negotiate the size of the data structures when the connection is established.
- Prefix variable-length data structures with the number of bytes being sent. The length value could be expressed either as a native integer value, or as a fixed-length string that is converted to a numeric value by the application. This allows the receiver to read this fixed length value, and then use that value to determine how many additional bytes must be read to obtain the complete message or data structure.
- Prefix the data with a unique byte or byte sequence that would normally not be found in the data stream. This would be followed by the data itself, with a complete message received when another unique byte sequence is encountered. Alternatively, a unique byte sequence could be used to terminate a message. This is the approach that many Internet application protocols use, such as FTP, SMTP and POP3. Commands are sent as one or more printable characters, terminated with a carriage-return (CR) and linefeed (LF) byte sequence that tells the remote host that a complete command has been received.
- A combination of the above methods, using unique byte sequences, the message length and even additional information such as a CRC-32 checksum or MD5 hash to validate the integrity of the data. This would effectively create an "envelope" around the data being exchanged,

and additional checks could be made to ensure that the data has been received and processed correctly.

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

It is important to keep in mind that all of this is only required if you decide to use the lower-level methods. SocketWrench also has a number of high-level methods which automatically handle the lower level network I/O for you. For example, the **StoreStream** method will read a data stream from the server and store the contents in a file specified by your application. When using the high-level methods, the details of how the data is read and processed is handled by SocketWrench and no additional coding is required on your part.

Event Handling

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

Asynchronous network events occur when a non-blocking connection is established and a network event occurs, such as a connection completing or data arriving from the remote host. Status events are used to indicate a change in status, such as a blocking operation being canceled or the progress of an operation such as reading a stream of bytes from the socket and storing it in a file. Note that asynchronous network events require that the **Blocking** property be set to **false**. The following events can be generated when SocketWrench is in non-blocking mode:

OnConnect

This event occurs whenever a connection to a remote host has completed. Unlike a blocking connection, when the component is in non-blocking mode, a successful call to the **Connect** method does not indicate that you are actually connected to the host. Instead, it means that the connection process has been started. Your application will not actually be connected until the **OnConnect** event fires.

OnDisconnect

This event occurs whenever the remote host closes its socket and terminates the connection with your application. Note that this event will not fire when you disconnect from the host by calling the **Disconnect** method; it only fires when the remote host closes its connection to you. It is also important to keep in mind that although the remote host has disconnected from you, there still may be data buffered on your local system, waiting to be read. If you are performing any low-level network I/O, your program should continue to call the **Read** method until it returns a value of zero, indicating that all of the available data has been read.

OnRead

This event occurs whenever the remote host sends data to your application. Once this event has fired, it will not be triggered again until you read at least one byte of data that has been sent to you. It is recommended that you attempt to read and buffer all of the data that is available to be read in the socket. When the **Read** method returns a value less than or equal to zero, you should exit the event handler.

OnWrite

This event occurs whenever there is enough memory available in the local send buffers to accommodate some data. It is generated immediately after a connection has completed, which tells your application that it may begin sending data to the remote host. It will also be generated if a call to the **Write** method fails with the error that it would cause the thread to block. In this case, when the socket is able to accept more data, the **OnWrite** event will fire.

An important consideration when it comes to event handling is that all asynchronous network events are level triggered. This means that once an event is fired, it will not be fired again until some action is taken by the application to handle the event. This is most commonly found with **OnRead** events, which are generated when the remote host sends data to your application, signaling to you that there is data available to be read. Even though the remote host may continue sending you more data, another **OnRead** event will not be generated until you read at least one byte of the data that has been sent to you. This is done to prevent the application from being flooded with event notifications. However, failure to handle an event can cause event notification to appear to stall. It is recommended that you do not do excessive processing in an event handler

that would cause the thread to block or enter a message loop. This can have a significant negative effect on performance and can lead to unexpected behavior on the part of your application. Instead, it's recommended that you buffer the data that you receive and then process that data after exiting the event handler.

Status related events are different because they do not depend on the value of the **Blocking** property, and are not directly related to asynchronous network operations. The most typical status event is the **OnProgress** event, which is used to provide information to the application about the status of a blocking operation, such as reading a data stream using the `ReadStream` method. The possible status events are:

OnCancel

This event is used by the class to indicate that a blocking network operation has been canceled by a call to the **Cancel** method. It is important to note that when the **Cancel** method is called, the blocking socket operation will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other blocking function. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls. The **OnCancel** event handler should only be used for notification purposes or updating the internal state of the application. It is not recommended that you perform any network operations inside this event handler.

OnError

This event is used by the class to indicate an error has occurred. This event is only generated when a method is called, never as the result of setting a property value. The **OnError** event handler should only be used for notification purposes or updating the internal state of the application. It is not recommended that you perform any network operations inside this event handler.

OnProgress

This event is used by the class to inform the application of the progress of a blocking operation, such as a file transfer. Note that in some cases, the class may not be able to determine the total amount of data to be transferred, which would prevent a percentage from being calculated. In this case, because the server is unable to specify the total size of the resource, the class will not be able to calculate a percentage. Instead, it will simply inform the program of the amount of data copied to the local host up to that point.

OnTimeout

This event is used by the class to indicate a blocking operation has timed-out. A timeout period is specified by setting the **Timeout** property to a value greater than zero. The **OnTimeout** event handler should only be used for notification purposes or updating the internal state of the application. It is not recommended that you perform any network operations inside this event handler.

Status events are typically used to update a user interface. For example, the **OnProgress** event may be used to update a **ProgressBar** control, or a warning message may be displayed if an **OnError** event occurs.

Error Handling

Error conditions can occur in one of two general circumstances, either when setting a property in the control or when calling a method. If the error occurs when setting a property, an exception will be generated which must be caught and handled by the application. Failure to do this will typically result in the program displaying an error message and then terminating. For example, in Visual Basic.NET, the Try..Catch statements can be used to establish an error handler.

Methods are a bit different in that errors can be handled in one of two ways. By default, when a method is called it will return a value that indicates success or failure. For those methods that return boolean values, a value of **true** indicates success and a value of **false** indicates failure. For methods that return numeric values, such as the **Read** and **Write** methods, a return value of zero or greater indicates success, and a return value of -1 indicates failure.

If you prefer to handle exceptions, rather than check return values for each method call, SocketWrench has a property called **ThrowError**. If this property is set to **true**, when a method fails it will throw an exception that must be caught by the application. In that case, if an error occurs without there being an exception handler in place, the application will typically terminate. The exception class will be of the type [SocketTools.SocketWrenchException](#) and the **ErrorCode** property will specify the error that generated the exception. Refer to the Technical Reference for more information about this exception class.

To determine the error code for the last error generated by that instance of the class, use the **LastError** property. To display a description of the error to the user, the **LastErrorString** property will can be used. This returns a string that describes the error which corresponds to the value of the **LastError** property. It is permitted to set the **LastError** property to a value of zero in order to clear the last error code. It is important to note that the last error code only has meaning if the previous method call indicates the operation has failed. If the previous operation was successful, the value of the last error code will be undefined and should not be used.

Debugging Facilities

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

To provide logging functionality for your application, you must redistribute the **SocketTools11.TraceLog.dll** library along with the class library. This library is what performs the actual logging and must be placed in the same folder where your application is installed. Note that you cannot add this library as a reference to your project. It is used internally by SocketWrench and cannot be used directly by your program.

To create a trace log, your application must set the **TraceFile** property to the name of a file, the **TraceFlags** property to the level of logging desired and then set the **Trace** property to **true**. The default level of logging, zero, specifies that general information about the function calls being made will be logged. The most detailed logging is provided by specifying a level of four. In that case, all data exchanged between your application and the remote host is logged. This provides the most information, however it also generates the largest log files. To disable logging, set the **Trace** property to false.

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

If your application uses multiple instances of the class, it is only necessary to enable logging in one of them. Once enabled, all network operations in the current thread will be logged, regardless of which instance has enabled logging.

InternetDialer Class

Implements an interface to the Remote Access Services API.

For a list of all members of this type, see [InternetDialer Members](#).

System.Object

SocketTools.InternetDialer

[Visual Basic]

Public Class InternetDialer

Implements IDisposable

[C#]

public class InternetDialer : IDisposable

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

The InternetDialer class provides a way for client applications to connect to a remote server using Microsoft Windows Remote Access Services (RAS). To use this class, the dial-up networking software must be installed on the local system. For access to the Internet, the TCP/IP protocol must be installed and configured. The class may be configured to use either the SLIP or PPP protocols, depending on the requirements of the service provider. Refer to your system documentation for information about installing and configuring dial-up networking on your system.

For those applications which may be used in a mobile environment, or otherwise require remote network access, the InternetDialer class provides a convenient interface to this service. Connections can be established and discontinued under the direct control of the program, rather than requiring that the user execute another program before starting your application.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[InternetDialer Members](#) | [SocketTools Namespace](#)

InternetDialer Members

[InternetDialer overview](#)

Public Static (Shared) Fields

 rasDeviceATM	
 rasDeviceFrameRelay	
 rasDeviceGeneric	
 rasDeviceIRDA	
 rasDeviceISDN	
 rasDeviceModem	
 rasDevicePad	
 rasDeviceParallel	
 rasDevicePPPoE	
 rasDeviceSerial	
 rasDeviceSonet	
 rasDeviceSW56	
 rasDeviceVPN	
 rasDeviceX25	

Public Instance Constructors

 InternetDialer Constructor	Initializes a new instance of the InternetDialer class.

Public Instance Fields

Connection	Gets the handle for a dial-up networking session.
DeviceEntry	Gets the name of the specified device entry.
NameServer	Gets and sets the IP addresses of the nameservers assigned to the current phonebook entry.
PhoneBookEntry	Gets the name for the specified phone book entry.

Public Instance Properties

AreaCode	Gets and sets the area code for the current phonebook entry.
AutoConnect	Automatically inherit connections established by another process.
AutoDial	Enable and disable autodialing on the local system.
AutoDisconnect	Automatically disconnect from the remote server.
Blocking	Gets and sets the blocking state of the class.
BytesIn	Gets the number of bytes that have been received by the dial-up networking device.

BytesOut	Gets the number of bytes that have been transmitted by the dial-up networking device.
Callback	Specifies that the remote server should call the system back.
CallbackNumber	Gets and sets the telephone number for the remote server to call back on.
Connections	Gets the number of active dial-up networking sessions.
ConnectSpeed	Gets the line speed for the current dial-up networking connection.
CountryCode	Gets and sets the country code for the current phonebook entry.
CountryName	Gets and sets the country name for the current phonebook entry.
DefaultGateway	Enable and disable the default gateway for IP packets through the dial-up adapter.
DeviceCount	Gets the number of dial-up networking devices available.
DeviceName	Gets and sets the device name for the current dial-up networking connection.
DeviceType	Gets and sets the device type for the current dial-up networking connection.
DynamicAddress	Enables and disables the use of dynamically allocated IP addresses.
DynamicNameServers	Enables and disables the use of dynamically assigned nameserver addresses.
EntryName	Gets and sets the current phone book entry name.
FramingProtocol	Gets and sets the framing protocol for the current phonebook entry.
Handle	Gets and sets the handle for the current dial-up networking connection.
InternetAddress	Gets the address assigned to the current dial-up networking session.
Interval	Gets and sets the interval at which the connection is monitored.
IpHeaderCompression	Enables and disables IP header compression for the current phonebook entry.
IsConnected	Gets a value which indicates if a connection has been established.
IsInitialized	Gets a value which indicates if the current instance of the class has been initialized successfully.
LastError	Gets and sets a value which specifies the last error

	that has occurred.
LastErrorString	Gets a value which describes the last error that has occurred.
LcpExtensions	Enables and disables the use of PPP LCP extensions for the current phonebook entry.
LocalNumber	Gets and sets the local telephone number specified in the phonebook entry.
ModemLights	Enables and disables the dial-up networking system tray icon.
ModemSpeaker	Enables and disables the modem speaker.
NetworkLogon	Enables and disables a network login for the current phonebook entry.
NetworkProtocol	Gets and sets the network protocol for the current phonebook entry.
Password	Gets the password required to establish a connection with the service provider.
PhoneBook	Sets the file name of the Remote Access phonebook to use.
PhoneBookEntries	Gets the number of entries in the current phonebook.
PhoneNumber	Gets and sets the telephone number of the service provider.
RequireEncryption	Enables and disables secure authentication for the current phonebook entry.
ScriptFile	Gets and sets the name of the script file for the current phonebook entry.
ServerAddress	Gets the address of the dial-up networking server.
SoftwareCompression	Enables and disables software compression for the current phonebook entry.
Status	Gets a value which specifies the current status of the dial-up networking connection.
Terminal	Gets and sets the interactive terminal window mode for the current phonebook entry.
ThrowError	Gets and sets a value which specifies if method calls should throw exceptions when an error occurs.
Timeout	Gets and sets a value which specifies a timeout period in seconds.
UserDomain	Gets and sets the NT domain on which user authentication is to occur.
UserName	Gets the username required to establish a connection with the service provider.
UserPhoneBook	Gets the name of the default user phonebook.

Version	Gets a value which returns the current version of the InternetDialer class library.

Public Instance Methods

Connect	Overloaded. Establish a connection with a dial-up networking services provider.
CreateEntry	Create a new entry in the current phonebook.
DeleteEntry	Overloaded. Delete a phonebook entry from the current phonebook.
Disconnect	Terminate the connection with the dial-up networking service provider.
Dispose	Overloaded. Releases all resources used by InternetDialer .
EditEntry	Edit an existing phonebook entry in the current phonebook.
Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
Initialize	Overloaded. Initialize an instance of the InternetDialer class.
LoadEntry	Overloaded. Load the specified entry from the current phonebook.
RenameEntry	Rename an existing phonebook entry.
Reset	Reset the internal state of the object, resetting all properties to their default values.
SaveEntry	Overloaded. Save the current settings to the specified phonebook entry in the current phonebook.
ToString (inherited from Object)	Returns a String that represents the current Object.
Uninitialize	Uninitialize the class library and release any resources allocated for the current thread.

Public Instance Events

OnCancel	Occurs when a blocking client operation is canceled.
OnConnect	Occurs when a connection is established with the service provider.
OnDisconnect	Occurs when the dial-up networking connection is terminated.
OnError	Occurs when an client operation fails.

OnStatus	Occurs when the when the connection state changes.
OnTimeout	Occurs when a blocking operation fails to complete before the timeout period elapses.

Protected Instance Methods

 Dispose	Overloaded. Releases the unmanaged resources allocated by the InternetDialer class and optionally releases the managed resources.
 Finalize	Destroys an instance of the class, releasing the resources allocated for the session and unloading the networking library.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer Constructor

Initializes a new instance of the InternetDialer class.

[Visual Basic]

```
Public Sub New()
```

[C#]

```
public InternetDialer();
```

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InternetDialer Fields

The fields of the **InternetDialer** class are listed below. For a complete list of **InternetDialer** class members, see the [InternetDialer Members](#) topic.

Public Static (Shared) Fields

 rasDeviceATM	
 rasDeviceFrameRelay	
 rasDeviceGeneric	
 rasDeviceIRDA	
 rasDeviceISDN	
 rasDeviceModem	
 rasDevicePad	
 rasDeviceParallel	
 rasDevicePPPoE	
 rasDeviceSerial	
 rasDeviceSonet	
 rasDeviceSW56	
 rasDeviceVPN	
 rasDeviceX25	

Public Instance Fields

Connection	Gets the handle for a dial-up networking session.
DeviceEntry	Gets the name of the specified device entry.
NameServer	Gets and sets the IP addresses of the nameservers assigned to the current phonebook entry.
PhoneBookEntry	Gets the name for the specified phone book entry.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Connection Field

Gets the handle for a dial-up networking session.

[Visual Basic]

Public **ReadOnly** **Connection** As [ConnectionArray](#)

[C#]

public **readonly** [ConnectionArray](#) **Connection**;

Remarks

The **Connection** array can be used to enumerate the active dial-up networking sessions on the local system. The index is zero-based, and the number of connections is returned by the **Connections** property. The property returns an integer value which represents the handle to the session. Setting the **Handle** property to this value will cause the control to inherit the session and the control's properties will be updated with information about the connection.

Specifying an index greater than the number of available connections will generate an exception.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.ConnectionArray Class

The class used to return the available connection handles.

For a list of all members of this type, see [InternetDialer.ConnectionArray Members](#).

System.Object

SocketTools.InternetDialer.ConnectionArray

[Visual Basic]

```
<DefaultMember(MemberName:="Item")>  
Public Class InternetDialer.ConnectionArray
```

[C#]

```
[DefaultMember(MemberName="Item")]  
public class InternetDialer.ConnectionArray
```

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

The **ConnectionArray** class returns the available dial-up networking connection handles. A read-only instance of this class is created by the InternetDialer class constructor and accessed using the **Connection** array

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[InternetDialer.ConnectionArray Members](#) | [SocketTools Namespace](#)

InternetDialer.ConnectionArray Members

[InternetDialer.ConnectionArray overview](#)

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Gets the handle for a dial-up networking connection.

Public Instance Methods

EntryName	Gets the phonebook entry name for the specified connection.
Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
Handle	Gets the handle for the specified dial-up networking connection.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetDialer.ConnectionArray Class](#) | [SocketTools Namespace](#)

InternetDialer.ConnectionArray Properties

The properties of the **InternetDialer.ConnectionArray** class are listed below. For a complete list of **InternetDialer.ConnectionArray** class members, see the [InternetDialer.ConnectionArray Members](#) topic.

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Gets the handle for a dial-up networking connection.

See Also

[InternetDialer.ConnectionArray Class](#) | [SocketTools Namespace](#)

InternetDialer.ConnectionArray.Count Property

Returns the maximum number of elements in the array.

[Visual Basic]

Public ReadOnly Property Count As Integer

[C#]

public int Count {get;}

Property Value

An integer value that specifies the number of active connections.

Remarks

This property will return the same value as the **Connections** property and is used to determine the maximum index value for the **Connection** array.

See Also

[InternetDialer.ConnectionArray Class](#) | [SocketTools Namespace](#)

InternetDialer.ConnectionArray.Item Property

Gets the handle for a dial-up networking connection.

[Visual Basic]

```
Public Default ReadOnly Property Item( _  
    ByVal index As Integer _  
) As Integer
```

[C#]

```
public int this[  
    int index  
] {get;}
```

Parameters

index

An integer value which specifies the index into the array.

Property Value

An integer value which specifies a dial-up networking connection handle.

See Also

[InternetDialer.ConnectionArray Class](#) | [SocketTools Namespace](#)

InternetDialer.ConnectionArray Methods

The methods of the **InternetDialer.ConnectionArray** class are listed below. For a complete list of **InternetDialer.ConnectionArray** class members, see the [InternetDialer.ConnectionArray Members](#) topic.

Public Instance Methods

EntryName	Gets the phonebook entry name for the specified connection.
Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
Handle	Gets the handle for the specified dial-up networking connection.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetDialer.ConnectionArray Class](#) | [SocketTools Namespace](#)

InternetDialer.ConnectionArray.EntryName Method

Gets the phonebook entry name for the specified connection.

[Visual Basic]

```
Public Function EntryName( _  
    ByVal index As Integer _  
) As String
```

[C#]

```
public string EntryName(  
    int index  
);
```

Parameters

index

An integer value which specifies the index into the array.

Return Value

A string which specifies the phonebook entry name for the active connection.

See Also

[InternetDialer.ConnectionArray Class](#) | [SocketTools Namespace](#)

InternetDialer.ConnectionArray.Handle Method

Gets the handle for the specified dial-up networking connection.

```
[Visual Basic]
Public Function Handle( _
    ByVal index As Integer _
) As Integer
```

```
[C#]
public int Handle(
    int index
);
```

Parameters

index

An integer value which specifies the index into the array.

Return Value

An integer value which specifies a dial-up networking connection handle.

See Also

[InternetDialer.ConnectionArray Class](#) | [SocketTools Namespace](#)

InternetDialer.DeviceEntry Field

Gets the name of the specified device entry.

[Visual Basic]

```
Public ReadOnly DeviceEntry As DeviceEntryArray
```

[C#]

```
public readonly DeviceEntryArray DeviceEntry;
```

Remarks

The **DeviceEntry** array can be used in conjunction with the **DeviceCount** property to enumerate the available dial-up networking devices. Typically this is used to provide a user with a selection of dial-up devices. The device used by the current phonebook entry can be changed by setting the **DeviceName** property to one of the device entry values.

Note that you should first set the **DeviceType** property to the type of device which you wish to enumerate. The default device type is **Modem**, for serial analog modems or other devices which recognize the AT command set.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.DeviceEntryArray Class

The class used to return the available dial-up networking devices.

For a list of all members of this type, see [InternetDialer.DeviceEntryArray Members](#).

System.Object

SocketTools.InternetDialer.DeviceEntryArray

[Visual Basic]

```
<DefaultMember(MemberName:="Item")>  
Public Class InternetDialer.DeviceEntryArray
```

[C#]

```
[DefaultMember(MemberName="Item")]  
public class InternetDialer.DeviceEntryArray
```

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

The **DeviceEntryArray** class returns the available dial-up networking devices. A read-only instance of this class is created by the InternetDialer class constructor and accessed using the **DeviceEntry** property array.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[InternetDialer.DeviceEntryArray Members](#) | [SocketTools Namespace](#)

InternetDialer.DeviceEntryArray Members

[InternetDialer.DeviceEntryArray overview](#)

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Gets the name of a device.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object .
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object .

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object .

See Also

[InternetDialer.DeviceEntryArray Class](#) | [SocketTools Namespace](#)

InternetDialer.DeviceEntryArray Properties

The properties of the **InternetDialer.DeviceEntryArray** class are listed below. For a complete list of **InternetDialer.DeviceEntryArray** class members, see the [InternetDialer.DeviceEntryArray Members](#) topic.

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Gets the name of a device.

See Also

[InternetDialer.DeviceEntryArray Class](#) | [SocketTools Namespace](#)

InternetDialer.DeviceEntryArray.Count Property

Returns the maximum number of elements in the array.

[Visual Basic]

Public ReadOnly Property Count As Integer

[C#]

public int Count {get;}

Property Value

An integer value that specifies the number of available devices.

Remarks

This property will return the same value as the **DeviceCount** property and is used to determine the maximum index value for the **DeviceEntry** array.

See Also

[InternetDialer.DeviceEntryArray Class](#) | [SocketTools Namespace](#)

InternetDialer.DeviceEntryArray.Item Property

Gets the name of a device.

[Visual Basic]

```
Public Default ReadOnly Property Item( _  
    ByVal index As Integer _  
) As String
```

[C#]

```
public string this[  
    int index  
] {get;}
```

Parameters

index

An integer value which specifies the index into the array.

Property Value

A string which specifies the device name.

See Also

[InternetDialer.DeviceEntryArray Class](#) | [SocketTools Namespace](#)

InternetDialer.NameServer Field

Gets and sets the IP addresses of the nameservers assigned to the current phonebook entry.

[Visual Basic]

```
Public ReadOnly NameServer As NameServerArray
```

[C#]

```
public readonly NameServerArray NameServer;
```

Remarks

The **NameServer** array is used to set or return the nameserver IP addresses assigned to the current phonebook entry. Setting the array to an IP address changes the corresponding address assigned to the phonebook entry. Note that assigned nameserver addresses are only used if the **DynamicNameServers** property has been set to **false**. If dynamic nameservers are assigned to the session this array will not return those addresses, it will return empty strings.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.NameServerArray Class

The class used to return the nameservers for the current phonebook entry.

For a list of all members of this type, see [InternetDialer.NameServerArray Members](#).

System.Object

SocketTools.InternetDialer.NameServerArray

[Visual Basic]

```
<DefaultMember(MemberName:="Item")>  
Public Class InternetDialer.NameServerArray
```

[C#]

```
[DefaultMember(MemberName="Item")]  
public class InternetDialer.NameServerArray
```

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

The **NameServerArray** class returns the nameservers for the current phonebook entry. A read-only instance of this class is created by the InternetDialer class constructor and accessed using the **NameServer** array.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[InternetDialer.NameServerArray Members](#) | [SocketTools Namespace](#)

InternetDialer.NameServerArray Members

[InternetDialer.NameServerArray overview](#)

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Gets the address of a nameserver.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetDialer.NameServerArray Class](#) | [SocketTools Namespace](#)

InternetDialer.NameServerArray Properties

The properties of the **InternetDialer.NameServerArray** class are listed below. For a complete list of **InternetDialer.NameServerArray** class members, see the [InternetDialer.NameServerArray Members](#) topic.

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Gets the address of a nameserver.

See Also

[InternetDialer.NameServerArray Class](#) | [SocketTools Namespace](#)

InternetDialer.NameServerArray.Count Property

Returns the maximum number of elements in the array.

[Visual Basic]

Public ReadOnly Property Count As Integer

[C#]

public int Count {get;}

Property Value

An integer value that specifies the maximum number of nameservers.

See Also

[InternetDialer.NameServerArray Class](#) | [SocketTools Namespace](#)

InternetDialer.NameServerArray.Item Property

Gets the address of a nameserver.

[Visual Basic]

```
Public Default Property Item( _  
    ByVal index As Integer _  
) As String
```

[C#]

```
public string this[  
    int index  
] {get; set;}
```

Parameters

index

An integer value which specifies the index into the array.

Property Value

A string which specifies the Internet Protocol (IP) address of a nameserver.

See Also

[InternetDialer.NameServerArray Class](#) | [SocketTools Namespace](#)

InternetDialer.PhoneBookEntry Field

Gets the name for the specified phone book entry.

[Visual Basic]

Public ReadOnly PhoneBookEntry As [PhoneBookEntryArray](#)

[C#]

public readonly [PhoneBookEntryArray](#) PhoneBookEntry;

Remarks

The **PhoneBookEntry** array contains a list of the entries in the current phone book, and may be used to establish a connection with a remote server. Specifying an index greater than the number of available entries in the phone book will generate an exception.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.PhoneBookEntryArray Class

The class used to return the available entries in the current phonebook.

For a list of all members of this type, see [InternetDialer.PhoneBookEntryArray Members](#).

System.Object

SocketTools.InternetDialer.PhoneBookEntryArray

[Visual Basic]

```
<DefaultMember(MemberName:="Item")>  
Public Class InternetDialer.PhoneBookEntryArray
```

[C#]

```
[DefaultMember(MemberName="Item")]  
public class InternetDialer.PhoneBookEntryArray
```

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

The **PhoneBookEntryArray** class returns the available entries in the current phonebook. A read-only instance of this class is created by the InternetDialer class constructor and accessed using the **PhoneBookEntry** array.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[InternetDialer.PhoneBookEntryArray Members](#) | [SocketTools Namespace](#)

InternetDialer.PhoneBookEntryArray Members

[InternetDialer.PhoneBookEntryArray overview](#)

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Gets the name of a phonebook entry.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetDialer.PhoneBookEntryArray Class](#) | [SocketTools Namespace](#)

InternetDialer.PhoneBookEntryArray Properties

The properties of the **InternetDialer.PhoneBookEntryArray** class are listed below. For a complete list of **InternetDialer.PhoneBookEntryArray** class members, see the [InternetDialer.PhoneBookEntryArray Members](#) topic.

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Gets the name of a phonebook entry.

See Also

[InternetDialer.PhoneBookEntryArray Class](#) | [SocketTools Namespace](#)

InternetDialer.PhoneBookEntryArray.Count Property

Returns the maximum number of elements in the array.

[Visual Basic]

Public ReadOnly Property Count As Integer

[C#]

public int Count {get;}

Property Value

An integer value which specifies the number of phonebook entries.

Remarks

This property will return the same value as the **PhoneBookEntries** property and is used to determine the maximum index value for the **PhoneBookEntry** array.

See Also

[InternetDialer.PhoneBookEntryArray Class](#) | [SocketTools Namespace](#)

InternetDialer.PhoneBookEntryArray.Item Property

Gets the name of a phonebook entry.

[Visual Basic]

```
Public Default ReadOnly Property Item( _  
    ByVal index As Integer _  
) As String
```

[C#]

```
public string this[  
    int index  
] {get;}
```

Parameters

index

An integer value which specifies the index into the array.

Property Value

A string which specifies the name of a phonebook entry.

See Also

[InternetDialer.PhoneBookEntryArray Class](#) | [SocketTools Namespace](#)

InternetDialer Properties

The properties of the **InternetDialer** class are listed below. For a complete list of **InternetDialer** class members, see the [InternetDialer Members](#) topic.

Public Instance Properties

AreaCode	Gets and sets the area code for the current phonebook entry.
AutoConnect	Automatically inherit connections established by another process.
AutoDial	Enable and disable autodialing on the local system.
AutoDisconnect	Automatically disconnect from the remote server.
Blocking	Gets and sets the blocking state of the class.
BytesIn	Gets the number of bytes that have been received by the dial-up networking device.
BytesOut	Gets the number of bytes that have been transmitted by the dial-up networking device.
Callback	Specifies that the remote server should call the system back.
CallbackNumber	Gets and sets the telephone number for the remote server to call back on.
Connections	Gets the number of active dial-up networking sessions.
ConnectSpeed	Gets the line speed for the current dial-up networking connection.
CountryCode	Gets and sets the country code for the current phonebook entry.
CountryName	Gets and sets the country name for the current phonebook entry.
DefaultGateway	Enable and disable the default gateway for IP packets through the dial-up adapter.
DeviceCount	Gets the number of dial-up networking devices available.
DeviceName	Gets and sets the device name for the current dial-up networking connection.
DeviceType	Gets and sets the device type for the current dial-up networking connection.
DynamicAddress	Enables and disables the use of dynamically allocated IP addresses.
DynamicNameServers	Enables and disables the use of dynamically assigned nameserver addresses.
EntryName	Gets and sets the current phone book entry name.

FramingProtocol	Gets and sets the framing protocol for the current phonebook entry.
Handle	Gets and sets the handle for the current dial-up networking connection.
InternetAddress	Gets the address assigned to the current dial-up networking session.
Interval	Gets and sets the interval at which the connection is monitored.
IpHeaderCompression	Enables and disables IP header compression for the current phonebook entry.
IsConnected	Gets a value which indicates if a connection has been established.
IsInitialized	Gets a value which indicates if the current instance of the class has been initialized successfully.
LastError	Gets and sets a value which specifies the last error that has occurred.
LastErrorString	Gets a value which describes the last error that has occurred.
LcpExtensions	Enables and disables the use of PPP LCP extensions for the current phonebook entry.
LocalNumber	Gets and sets the local telephone number specified in the phonebook entry.
ModemLights	Enables and disables the dial-up networking system tray icon.
ModemSpeaker	Enables and disables the modem speaker.
NetworkLogon	Enables and disables a network login for the current phonebook entry.
NetworkProtocol	Gets and sets the network protocol for the current phonebook entry.
Password	Gets the password required to establish a connection with the service provider.
PhoneBook	Sets the file name of the Remote Access phonebook to use.
PhoneBookEntries	Gets the number of entries in the current phonebook.
PhoneNumber	Gets and sets the telephone number of the service provider.
RequireEncryption	Enables and disables secure authentication for the current phonebook entry.
ScriptFile	Gets and sets the name of the script file for the current phonebook entry.
ServerAddress	Gets the address of the dial-up networking server.

SoftwareCompression	Enables and disables software compression for the current phonebook entry.
Status	Gets a value which specifies the current status of the dial-up networking connection.
Terminal	Gets and sets the interactive terminal window mode for the current phonebook entry.
ThrowError	Gets and sets a value which specifies if method calls should throw exceptions when an error occurs.
Timeout	Gets and sets a value which specifies a timeout period in seconds.
UserDomain	Gets and sets the NT domain on which user authentication is to occur.
UserName	Gets the username required to establish a connection with the service provider.
UserPhoneBook	Gets the name of the default user phonebook.
Version	Gets a value which returns the current version of the InternetDialer class library.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.AreaCode Property

Gets and sets the area code for the current phonebook entry.

[Visual Basic]

```
Public Property AreaCode As String
```

[C#]

```
public string AreaCode {get; set;}
```

Property Value

A string which specifies the area code.

Remarks

The **AreaCode** property is used to set or return the current phonebook entry's area code. If no area code has been specified, then this property will return an empty string. The value of this property is ignored unless the **CountryCode** property is also set to a valid country code.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.AutoConnect Property

Automatically inherit connections established by another process.

[Visual Basic]

```
Public Property AutoConnect As Boolean
```

[C#]

```
public bool AutoConnect {get; set;}
```

Property Value

A boolean value which specifies if connections are automatically inherited by the class.

Remarks

The **AutoConnect** property determines if the class automatically detects if a connection has been established by another process. When enabled, the class will periodically check for any connections that have been established. The **Interval** property controls the frequency in which the control performs this check.

If the class detects that a connection has been made, it will immediately fire the **OnConnect** event, followed by the **OnStatus** event, to indicate that a connection has been established. The class then begins to monitor that connection as usual, until that connection is dropped or the control is unloaded.

To periodically check to see if a connection has been established by another process without using the **AutoConnect** property, read the value of the **Connections** property, which returns the number of active dial-up networking connections. A value greater than zero indicates that a dial-up networking connection has been established.

If there are multiple dial-up networking devices on the system, it may be possible for more than one connection to be active at a time. If this is the case, setting the **AutoConnect** property to **true** will cause the class to inherit the first active connection. To manage multiple dial-up connections, use the **Connection** array to enumerate the available connections and set the **Handle** property to take control of a specific session.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.AutoDial Property

Enable and disable autodialing on the local system.

[Visual Basic]

Public Property AutoDial As Boolean

[C#]

public bool AutoDial {get; set;}

Property Value

A boolean value which specifies if autodialing has been enabled.

Remarks

The **AutoDial** property can be used to determine if autodialing is enabled or disabled on the current system. When autodialing is enabled and an application attempts to establish a connection over the Internet, a dialog box will be displayed asking the user if they want to connect to their default service provider. This property will return **true** if autodialing is currently enabled, or **false** if it has been disabled.

Setting the **AutoDial** property allows an application to change the autodial settings for the current user. Setting the property value to **true** specifies that you wish to enable autodialing, and the system will prompt the user to establish a dial-up connection when necessary. Setting the property to **false** disables autodialing, and prevents the system from prompting the user. This can be beneficial if your application needs to run in an unattended mode. If you change the autodial settings for the user, it is recommended that you restore them to their original value before the application terminates.

If the autodial settings cannot be changed by the current user, an exception will be generated.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.AutoDisconnect Property

Automatically disconnect from the remote server.

[Visual Basic]

Public Property AutoDisconnect As Boolean

[C#]

public bool AutoDisconnect {get; set;}

Property Value

A boolean value that specifies if connections are automatically terminated.

Remarks

The **AutoDisconnect** property determines if this instance of the class should automatically disconnect from a remote host when the destructor is called, typically when the application terminates. The default value for this property is **true**.

If a dial-up connection was already established at the time an instance of the class is created, this property will be reset to **false**, preventing it from automatically disconnecting from the host when it is unloaded. Therefore, to always force the control to automatically terminate a connection when it is unloaded, you must explicitly set the property value to **true** in your application.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Blocking Property

Gets and sets the blocking state of the class.

[Visual Basic]

Public Property Blocking As Boolean

[C#]

public bool Blocking {get; set;}

Property Value

A boolean value which specifies the blocking state of the class.

Remarks

The **Blocking** property determines how the class establishes a dial-up connection. If set to **true**, the class will wait until a connection has been established or the connection attempt fails before returning control to the application. If set to **false**, the class will begin the connection process and return control immediately to the application. For a non-blocking connection, the application should monitor the **OnStatus** event to determine the progress of the connection attempt. The default value for this property is **false**.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.BytesIn Property

Gets the number of bytes that have been received by the dial-up networking device.

[Visual Basic]

```
Public ReadOnly Property BytesIn As Integer
```

[C#]

```
public int BytesIn {get;}
```

Property Value

An integer which specifies the number of bytes received.

Remarks

The **BytesIn** property returns the number of bytes that have been received by the dial-up networking device. If the control is unable to determine the number of bytes received, it will return a value of zero.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.BytesOut Property

Gets the number of bytes that have been transmitted by the dial-up networking device.

[Visual Basic]

```
Public ReadOnly Property BytesOut As Integer
```

[C#]

```
public int BytesOut {get;}
```

Property Value

An integer which specifies the number of bytes transmitted.

Remarks

The **BytesOut** property returns the number of bytes that have been transmitted by the dial-up networking device. If the control is unable to determine the number of bytes transmitted, it will return a value of zero.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Callback Property

Specifies that the remote server should call the system back.

[Visual Basic]

Public Property Callback As Boolean

[C#]

public bool Callback {get; set;}

Property Value

A boolean value which specifies if the server should call the local system back.

Remarks

Setting the **Callback** property specifies that the server should call the user back at the telephone number specified by the **CallbackNumber** property. This property is ignored unless the user has "Set By Caller" callback permission on the server.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.CallbackNumber Property

Gets and sets the telephone number for the remote server to call back on.

[Visual Basic]

Public Property CallbackNumber As String

[C#]

public string CallbackNumber {get; set;}

Property Value

A string which specifies the callback telephone number.

Remarks

Setting the **CallbackNumber** property specifies that the server should call the user back at the given telephone number. This property is ignored unless the user has "Set By Caller" callback permission on the server. Assigning an asterisk to this property causes the number stored in the phone book entry to be used for callback.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Connections Property

Gets the number of active dial-up networking sessions.

[Visual Basic]

Public ReadOnly Property Connections As Integer

[C#]

public int Connections {get;}

Property Value

An integer value which specifies the number of active dial-up networking sessions.

Remarks

The **Connections** property returns the number of active dial-up networking connections on the local system. A value of zero indicates that there is no dial-up networking connection. This property is used in conjunction with the **Connection** array to enumerate the connections on the current system.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.ConnectSpeed Property

Gets the line speed for the current dial-up networking connection.

[Visual Basic]

Public ReadOnly Property ConnectSpeed As Integer

[C#]

public int ConnectSpeed {get;}

Property Value

An integer value which specifies the connection speed.

Remarks

The **ConnectSpeed** property returns the speed, in bytes per second, at which the current dial-up networking device has established a connection. If the class is unable to determine the connection speed, it will return a value of zero.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.CountryCode Property

Gets and sets the country code for the current phonebook entry.

[Visual Basic]

Public Property CountryCode As Integer

[C#]

public int CountryCode {get; set;}

Property Value

An integer value which specifies the country code.

Remarks

The **CountryCode** property specifies the numeric country code for the current phonebook entry. If this value is zero, then the country and area code information is not used when dialing the phone number. The country code for the United States is 1.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.CountryName Property

Gets and sets the country name for the current phonebook entry.

[Visual Basic]

Public Property CountryName As String

[C#]

public string CountryName {get; set;}

Property Value

A string which specifies the country name.

Remarks

The **CountryName** property returns the name of the country associated with the country code used when dialing the current phonebook entry. If no country code has been specified, this property will return an empty string. Setting this property to the name of a country will change the current country code. If no area code has been defined, and the country code specifies the current dialing location, the **AreaCode** property will be updated to the current area code.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.DefaultGateway Property

Enable and disable the default gateway for IP packets through the dial-up adapter.

[Visual Basic]

Public Property DefaultGateway As Boolean

[C#]

public bool DefaultGateway {get; set;}

Property Value

A boolean value which specifies if the default gateway should be used.

Remarks

The **DefaultGateway** property is used to determine the default gateway for IP packets. If set to **true**, then packets are routed through the dial-up networking adapter when the connection is active. The value of this property corresponds to the Use Default Gateway checkbox on the TCP/IP configuration dialog.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.DeviceCount Property

Gets the number of dial-up networking devices available.

[Visual Basic]

```
Public ReadOnly Property DeviceCount As Integer
```

[C#]

```
public int DeviceCount {get;}
```

Property Value

An integer value which specifies the number of devices.

Remarks

The **DeviceCount** property returns the number of dial-up networking devices available. This property can be used in conjunction with the **DeviceEntry** array to enumerate the devices.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.DeviceName Property

Gets and sets the device name for the current dial-up networking connection.

[Visual Basic]

```
Public Property DeviceName As String
```

[C#]

```
public string DeviceName {get; set;}
```

Property Value

A string which specifies the device name.

Remarks

The **DeviceName** property returns a description of the device that the connection was established on. For example, the string "US Robotics Sportster 56000" may be returned for a modem. Note that this property value may change if the **DeviceType** property is modified. Setting this property will change the device used to establish the dial-up networking connection. Changes to this property value should be made after changes to the **DeviceType** property.

To enumerate a list of available devices for a given device type, use the **DeviceCount** property and **DeviceEntry** array.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.DeviceType Property

Gets and sets the device type for the current dial-up networking connection.

```
[Visual Basic]
Public Property DeviceType As String
```

```
[C#]
public string DeviceType {get; set;}
```

Property Value

A string which specifies the device type.

Remarks

The **DeviceType** property returns the type of device that the connection was established with. Setting this property will change the type of device that will be used to establish the connection. Examples of valid device types are:

DeviceType	Description
modem	An internal or external serial analog modem device, or other serial communications device which supports the AT command set.
isdn	An ISDN terminal adapter. Note that some ISDN devices, such as the 3Com ImpactIQ are considered modem devices.
vpn	A virtual private network connection.

Because changing the device type can change the current device name, it is recommended that applications change this property value before changing the value of the **DeviceName** property.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.DynamicAddress Property

Enables and disables the use of dynamically allocated IP addresses.

[Visual Basic]

Public Property DynamicAddress As Boolean

[C#]

public bool DynamicAddress {get; set;}

Property Value

A boolean value which specifies if a dynamically allocated IP address should be used.

Remarks

The **DynamicAddress** property determines if the current phonebook entry should use a dynamically assigned IP address. If this property is set to **true**, then an IP address is assigned to the dial-up adapter when the connection is established. If set to **false**, then the dial-up adapter IP address is set to the value of the **InternetAddress** property.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.DynamicNameServers Property

Enables and disables the use of dynamically assigned nameserver addresses.

[Visual Basic]

Public Property DynamicNameServers As Boolean

[C#]

public bool DynamicNameServers {get; set;}

Property Value

A boolean value which specifies if dynamically allocated nameservers should be used.

Remarks

The **DynamicNameServers** property determines if the current phonebook entry should use dynamically assigned nameservers. If this property is set to **true**, then one or more nameservers are assigned to the dial-up adapter when the connection is established. If set to **false**, then the dial-up adapter nameservers are set to the values specified by the **NameServer** property array.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.EntryName Property

Gets and sets the current phone book entry name.

[Visual Basic]

```
Public Property EntryName As String
```

[C#]

```
public string EntryName {get; set;}
```

Property Value

A string which specifies the current phonebook entry name.

Remarks

The **EntryName** property can be used to specify a phone book entry to use to connect with a remote server. The entry name identifies a communications profile which includes the telephone number, callback number and domain name of the remote host. Setting the **EntryName** property to an empty string indicates that a telephone number will be provided to establish the connection.

In Windows documentation, the phonebook entry name is also referred to as a *connectoid*.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.FramingProtocol Property

Gets and sets the framing protocol for the current phonebook entry.

[Visual Basic]

Public Property FramingProtocol As [RasFramingProtocol](#)

[C#]

public [InternetDialer.RasFramingProtocol](#) FramingProtocol {get; set;}

Property Value

A [RasFramingProtocol](#) enumeration which specifies the framing protocol.

Remarks

The **FramingProtocol** property is used to set or return the framing protocol used for the current phonebook entry.

Note that unless there is a specific need for the application to use SLIP or the Microsoft RAS protocol, it is recommended that PPP always be selected as the framing protocol.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Handle Property

Gets and sets the handle for the current dial-up networking connection.

[Visual Basic]

Public Property Handle As Integer

[C#]

public int Handle {get; set;}

Property Value

An integer value which specifies a dial-up networking connection.

Remarks

The **Handle** property returns the handle to the current dial-up networking connection, or a value of zero if the class has not been used to establish a connection. Setting the value of this property to a valid handle causes the class to inherit the specified connection, and its properties will be updated with information about that connection. This enables an application to monitor and control a connection that was established by another program.

Setting the **Handle** property to a value of zero causes the class to release the current connection, however it will not cause the dial-up networking session to terminate. To disconnect from the remote server, the **Disconnect** method must be called by the application. Setting the property to a non-zero value which does not specify a valid handle will generate an exception.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.InternetAddress Property

Gets the address assigned to the current dial-up networking session.

[Visual Basic]

```
Public Property InternetAddress As String
```

[C#]

```
public string InternetAddress {get; set;}
```

Property Value

A string which specifies an Internet Protocol (IP) address.

Remarks

The **InternetAddress** property returns the address assigned to the current dial-up networking session. If no connection has been established, or the connection has not been made with a PPP server, then this property will return an empty string. If the **DynamicAddress** property is set to **false**, changing this property value will update the address assigned to the current phonebook entry.

The address may only be changed before a connection is established.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Interval Property

Gets and sets the interval at which the connection is monitored.

[Visual Basic]

Public Property Interval As Integer

[C#]

public int Interval {get; set;}

Property Value

An integer value which specifies the interval in milliseconds.

Remarks

The **Interval** property specifies the interval, in milliseconds, at which the connection is monitored by the class. The minimum value of 0 indicates that the class should not monitor the connection. The maximum interval value is 65536 milliseconds, which is slightly more than one minute. The default value is 1000, which causes the control to check the connection status every second.

Note that setting the property value to zero will prevent the class from detecting certain conditions, such as a disconnected telephone line or a modem that is turned off.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.IpHeaderCompression Property

Enables and disables IP header compression for the current phonebook entry.

[Visual Basic]

Public Property IpHeaderCompression As Boolean

[C#]

public bool IpHeaderCompression {get; set;}

Property Value

A boolean value which specifies if IP header compression is enabled.

Remarks

The **IpHeaderCompression** property is used to enable or disable IP header compression. If set to **true**, when a connection is established, RAS will negotiate with the dial-up server to use header compression. If set to **false**, header compression will not be negotiated. This property corresponds to the Use IP Header Compression checkbox on the TCP/IP configuration dialog.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.IsConnected Property

Gets a value which indicates if a connection has been established.

[Visual Basic]

Public ReadOnly Property IsConnected As Boolean

[C#]

public bool IsConnected {get;}

Property Value

A boolean value which specifies if a connection has been established with a service provider.

Remarks

The **IsConnected** property is used to determine if the class has connected to the remote host. A value of **true** indicates that a connection has been established.

Note that the **IsConnected** property should not be used to determine if an active dial-up networking connection has been established by another application. The property will only return **true** if the class has been used to establish the connection itself, or if a connection is inherited by setting either the **AutoConnect** or **Handle** properties. To determine if there are any active dial-up networking connections, check the value of the **Connections** property.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.IsInitialized Property

Gets a value which indicates if the current instance of the class has been initialized successfully.

[Visual Basic]

Public ReadOnly Property IsInitialized As Boolean

[C#]

public bool IsInitialized {get;}

Property Value

Returns **true** if the class instance has been initialized; otherwise returns **false**.

Remarks

The **IsInitialized** property is used to determine if the current instance of the class has been initialized properly. Normally this is done automatically by the class constructor, however there are circumstances where the class may not be able to initialize itself.

The most common reasons that a class instance may not initialize correctly is that no runtime license key has been defined in the assembly or the license key provided is invalid. It may also indicate a problem with the system configuration or user access rights, such as not being able to load the required networking libraries or not being able to access the system registry.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.LastError Property

Gets and sets a value which specifies the last error that has occurred.

[Visual Basic]

Public Property LastError As [ErrorCode](#)

[C#]

public [InternetDialer.ErrorCode](#) LastError {get; set;}

Property Value

Returns an [ErrorCode](#) enumeration value which specifies the last error code.

Remarks

The **LastError** property returns the error code associated with the last error that occurred for the current instance of the class. It is important to note that this value only has meaning if the previous method indicates that an error has actually occurred.

It is possible to explicitly clear the last error code by assigning the property to the value **ErrorCode.None**.

The error code value can be cast to an integer value for display purposes if required. For a description of the error that can be displayed using a message box or some other similar mechanism, get the value of the **LastErrorString** property.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.LastErrorString Property

Gets a value which describes the last error that has occurred.

[Visual Basic]

```
Public ReadOnly Property LastErrorString As String
```

[C#]

```
public string LastErrorString {get;}
```

Property Value

A string which describes the last error that has occurred.

Remarks

The **LastErrorString** property can be used to obtain a description of the last error that occurred for the current instance of the class. It is important to note that this value only has meaning if the previous method indicates that an error has actually occurred.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.LcpExtensions Property

Enables and disables the use of PPP LCP extensions for the current phonebook entry.

[Visual Basic]

Public Property LcpExtensions As Boolean

[C#]

public bool LcpExtensions {get; set;}

Property Value

A boolean value which specifies if PPP LCP extensions are enabled.

Remarks

The **LcpExtensions** property determines if the PPP LCP extensions defined in RFC 1570 will be used. If the PPP framing protocol is being used for the dial-up connection, it is recommended that this property be set to **true**. However, some older implementations of PPP may require that this property be set to **false** in order to establish a connection.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.LocalNumber Property

Gets and sets the local telephone number specified in the phonebook entry.

[Visual Basic]

```
Public Property LocalNumber As String
```

[C#]

```
public string LocalNumber {get; set;}
```

Property Value

A string which specifies the local telephone number.

Remarks

The **LocalNumber** property sets or returns the local phone number that is specified in the current phonebook entry. If the **CountryCode** property has a value of zero, then the local number is dialed to connect to the server. If the **CountryCode** property is set to a valid country code, then RAS will also use the country and area code values when dialing the phone number.

Note that this property only determines the local phone number for the phonebook entry, and can be overridden by setting the **PhoneNumber** property to a specific value.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.ModemLights Property

Enables and disables the dial-up networking system tray icon.

[Visual Basic]

Public Property ModemLights As Boolean

[C#]

public bool ModemLights {get; set;}

Property Value

A boolean value which specifies if the system tray icon is displayed.

Remarks

The **ModemLights** property determines if the dial-up networking icon in the system tray is displayed when a connection is established.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.ModemSpeaker Property

Enables and disables the modem speaker.

[Visual Basic]

Public Property ModemSpeaker As Boolean

[C#]

public bool ModemSpeaker {get; set;}

Property Value

A boolean value which specifies if the modem speaker is enabled.

Remarks

The **ModemSpeaker** property determines if the modem speaker is enabled when dialing the remote server. If the property is set to **false**, the modem will be silent when dialing the telephone number and establishing the connection. Note that setting this property to **true** will not force the speaker on if the modem hardware has been configured to explicitly disable the speaker.

To disable the speaker, the modem must support changes to the speaker volume. Disabling the speaker is typically done by instructing the modem to set the speaker volume to zero.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.NetworkLogon Property

Enables and disables a network login for the current phonebook entry.

[Visual Basic]

Public Property NetworkLogon As Boolean

[C#]

public bool NetworkLogon {get; set;}

Property Value

A boolean value which specifies if a network login is enabled.

Remarks

The **NetworkLogon** property determines if the client automatically logs on to the network after a connection has been established.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.NetworkProtocol Property

Gets and sets the network protocol for the current phonebook entry.

[Visual Basic]

Public Property NetworkProtocol As [RasNetworkProtocol](#)

[C#]

public [InternetDialer.RasNetworkProtocol](#) NetworkProtocol {get; set;}

Property Value

A [RasNetworkProtocol](#) enumeration value which specifies the network protocol.

Remarks

The **NetworkProtocol** property is used to set or return the network protocol used for the current phonebook entry.

Note that unless there is a specific need for the application to use the NetBEUI or IPX protocols, it is recommended that only the TCP/IP protocol be specified.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Password Property

Gets the password required to establish a connection with the service provider.

[Visual Basic]

```
Public Property Password As String
```

[C#]

```
public string Password {get; set;}
```

Property Value

A string which specifies the password for the current phonebook entry.

Remarks

The **Password** property specifies the password required to establish a connection with the service provider.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.PhoneBook Property

Sets the file name of the Remote Access phonebook to use.

[Visual Basic]

```
Public Property PhoneBook As String
```

[C#]

```
public string PhoneBook {get; set;}
```

Property Value

A string which specifies the current phonebook.

Remarks

The **PhoneBook** property specifies the file name of the Remote Access phone book. Setting this property to an empty string causes the default phonebook to be used.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.PhoneBookEntries Property

Gets the number of entries in the current phonebook.

[Visual Basic]

Public ReadOnly Property PhoneBookEntries As Integer

[C#]

public int PhoneBookEntries {get;}

Property Value

An integer value which specifies the number of phonebook entries.

Remarks

The **PhoneBookEntries** property returns the number of entries in the current phonebook. A value of zero indicates that no phonebook entries are available.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.PhoneNumber Property

Gets and sets the telephone number of the service provider.

[Visual Basic]

```
Public Property PhoneNumber As String
```

[C#]

```
public string PhoneNumber {get; set;}
```

Property Value

A string which specifies a telephone number.

Remarks

The **PhoneNumber** property specifies the telephone number of the service provider. If this property is not set, then the **PhoneEntry** property must be set to a valid phone book entry. If both the **PhoneNumber** and **PhoneEntry** properties are defined, the **PhoneNumber** property will override the value specified in the phonebook.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.RequireEncryption Property

Enables and disables secure authentication for the current phonebook entry.

[Visual Basic]

Public Property RequireEncryption As Boolean

[C#]

public bool RequireEncryption {get; set;}

Property Value

A boolean value which specifies if secure authentication is enabled.

Remarks

The **RequireEncryption** property determines if encryption is required during PPP authentication. If the property is set to **true**, then only secure password schemes can be used to authenticate the client. If the property is set to **false**, the client can use the PAP plain-text authentication protocol to authenticate the client. Some older PPP implementations may require that this property be set to **false** in order to establish a connection.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.ScriptFile Property

Gets and sets the name of the script file for the current phonebook entry.

[Visual Basic]

Public Property ScriptFile As String

[C#]

public string ScriptFile {get; set;}

Property Value

A string which specifies the name of the script file.

Remarks

The **ScriptFile** property specifies the name of the login script used to establish a connection with the remote host. This property must be set to the full pathname of the script file. If a script file is not required, then this property should be set to an empty string.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.ServerAddress Property

Gets the address of the dial-up networking server.

[Visual Basic]

```
Public ReadOnly Property ServerAddress As String
```

[C#]

```
public string ServerAddress {get;}
```

Property Value

A string which specifies an Internet Protocol (IP) address.

Remarks

The **ServerAddress** property returns the address of the dial-up networking server that the local host has connected to. If no connection has been established, or the connection has not been made with a PPP server, then this property will return an empty string. This property may also return an empty string if the remote server did not provide this information during the connection process.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.SoftwareCompression Property

Enables and disables software compression for the current phonebook entry.

[Visual Basic]

Public Property SoftwareCompression As Boolean

[C#]

public bool SoftwareCompression {get; set;}

Property Value

A boolean value which specifies if software compression is enabled.

Remarks

The **SoftwareCompression** property determines if data compression is negotiated during the connection. If the property is set to **true**, then the client will negotiate a compatible compression protocol. Software compression should only be disabled if the client is unable to establish a connection with the server.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Status Property

Gets a value which specifies the current status of the dial-up networking connection.

[Visual Basic]

Public **ReadOnly** **Property** **Status** **As** [DialerStatus](#)

[C#]

public [InternetDialer.DialerStatus](#) **Status** {get;}

Property Value

A [DialerStatus](#) enumeration value which specifies the current status.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Terminal Property

Gets and sets the interactive terminal window mode for the current phonebook entry.

[Visual Basic]

Public Property Terminal As [RasTerminalMode](#)

[C#]

public [InternetDialer.RasTerminalMode](#) Terminal {get; set;}

Property Value

A [RasTerminalMode](#) enumeration which specifies the terminal window mode.

Remarks

The **Terminal** array is used to control if a terminal window is displayed during the dial-up networking connection process. The terminal window can be used to allow user input before and/or after the dial-up networking connection has been established. If scripting has been enabled by setting the **ScriptFile** property, no terminal window should be displayed after the connection. This is because scripting has it's own terminal implementation.

Displaying a terminal window also imposes several restrictions on the behavior of the class. Because of how the Remote Access Services API is implemented by Microsoft, a connection dialog will be displayed after the **Connect** method is called if the **Terminal** property is non-zero. Setting this property to a non-zero value will also disable any asynchronous event notifications. It is not recommended that you set this property unless it is absolutely necessary.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.ThrowError Property

Gets and sets a value which specifies if method calls should throw exceptions when an error occurs.

[Visual Basic]

Public Property ThrowError As Boolean

[C#]

public bool ThrowError {get; set;}

Property Value

Returns **true** if method calls will generate exceptions when an error occurs; otherwise returns **false**. The default value is **false**.

Remarks

Error handling for when calling class methods can be done in either of two different styles, according to the value of this property.

If the **ThrowError** property is set to **false**, the application should check the return value of any method that is used, and report errors based upon the documented value of the return code. It is the responsibility of the application to interpret the error code, if it is desired to explain the error in addition to reporting it. This is the default behavior.

If the **ThrowError** property is set to **true**, then exceptions will be generated whenever a method call fails. The program must be written to catch these exceptions and take the appropriate action when an error occurs. Failure to handle an exception will cause the program to terminate abnormally.

Note that if an error occurs while a property is being read or modified, an exception will be raised regardless of the value of the **ThrowError** property.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Timeout Property

Gets and sets a value which specifies a timeout period in seconds.

[Visual Basic]

Public Property Timeout As Integer

[C#]

public int Timeout {get; set;}

Property Value

An integer value which specifies a timeout period in seconds.

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and returns an error. The timeout period is only used when the client is in blocking mode. Although this property can be changed when the client is in non-blocking mode, the value will be ignored until the client is returned to blocking mode.

For most applications it is recommended the timeout period be set between 10 and 20 seconds.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.UserDomain Property

Gets and sets the NT domain on which user authentication is to occur.

[Visual Basic]

```
Public Property UserDomain As String
```

[C#]

```
public string UserDomain {get; set;}
```

Property Value

A string which specifies the NT domain name.

Remarks

The **UserDomain** property is used to specify the NT domain on which the user name and password will be authenticated. An empty string specifies the domain in which the Remote Access server is a member. An asterisk specifies the domain stored in the phone book entry.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.UserName Property

Gets the username required to establish a connection with the service provider.

[Visual Basic]

```
Public Property UserName As String
```

[C#]

```
public string UserName {get; set;}
```

Property Value

A string which specifies the username for the current phonebook entry.

Remarks

The **UserName** property specifies the username required to establish a connection with the service provider.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.UserPhoneBook Property

Gets the name of the default user phonebook.

[Visual Basic]

```
Public ReadOnly Property UserPhoneBook As String
```

[C#]

```
public string UserPhoneBook {get;}
```

Property Value

A string which specifies a phonebook name.

Remarks

The **UserPhoneBook** property returns the name of the default user phonebook. The value returned depends on how the user has configured dial-up networking, specifically whether the system, user or alternate phonebook has been selected. The current phonebook can be changed by setting the **PhoneBook** property.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Version Property

Gets a value which returns the current version of the InternetDialer class library.

[Visual Basic]

Public ReadOnly Property Version As String

[C#]

public string Version {get;}

Property Value

A string which specifies the version of the class library.

Remarks

The **Version** property returns a string which identifies the current version and build of the InternetDialer class library. This value can be used by an application for validation and debugging purposes.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer Methods

The methods of the **InternetDialer** class are listed below. For a complete list of **InternetDialer** class members, see the [InternetDialer Members](#) topic.

Public Instance Methods

Connect	Overloaded. Establish a connection with a dial-up networking services provider.
CreateEntry	Create a new entry in the current phonebook.
DeleteEntry	Overloaded. Delete a phonebook entry from the current phonebook.
Disconnect	Terminate the connection with the dial-up networking service provider.
Dispose	Overloaded. Releases all resources used by InternetDialer .
EditEntry	Edit an existing phonebook entry in the current phonebook.
Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
Initialize	Overloaded. Initialize an instance of the InternetDialer class.
LoadEntry	Overloaded. Load the specified entry from the current phonebook.
RenameEntry	Rename an existing phonebook entry.
Reset	Reset the internal state of the object, resetting all properties to their default values.
SaveEntry	Overloaded. Save the current settings to the specified phonebook entry in the current phonebook.
ToString (inherited from Object)	Returns a String that represents the current Object.
Uninitialize	Uninitialize the class library and release any resources allocated for the current thread.

Protected Instance Methods

 Dispose	Overloaded. Releases the unmanaged resources allocated by the InternetDialer class and optionally releases the managed resources.
 Finalize	Destroys an instance of the class, releasing the resources allocated for the session and unloading

	the networking library.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Connect Method

Establish a connection with a dial-up networking services provider.

Overload List

Establish a connection with a dial-up networking services provider.

```
public bool Connect();
```

Establish a connection with a dial-up networking services provider.

```
public bool Connect(string);
```

Establish a connection with a dial-up networking services provider.

```
public bool Connect(string,string,string);
```

Establish a connection with a dial-up networking services provider.

```
public bool Connect(string,string,string,string);
```

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Connect Method ()

Establish a connection with a dial-up networking services provider.

[Visual Basic]

Overloads Public Function Connect() As Boolean

[C#]

public bool Connect();

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The **Connect** method establishes a dial-up networking connection with a service provider using the current phonebook entry.

The current phonebook entry name is specified by the **EntryName** property.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [InternetDialer.Connect Overload List](#)

InternetDialer.Connect Method (String)

Establish a connection with a dial-up networking services provider.

```
[Visual Basic]
Overloads Public Function Connect( _
    ByVal entryName As String _
) As Boolean
```

```
[C#]
public bool Connect(
    string entryName
);
```

Parameters

entryName

A string which specifies the phonebook entry that should be used when establishing the connection.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The **Connect** method establishes a dial-up networking connection with a service provider using the specified phonebook entry. The entry name is the same name as the connectoid that is displayed when you list the available dial-up networking connections on the local system.

For a list of all of the available phonebook entries, reference the **PhoneBookEntry** array.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [InternetDialer.Connect Overload List](#)

InternetDialer.Connect Method (String, String, String)

Establish a connection with a dial-up networking services provider.

[Visual Basic]

```
Overloads Public Function Connect( _  
    ByVal phoneNumber As String, _  
    ByVal userName As String, _  
    ByVal userPassword As String _  
) As Boolean
```

[C#]

```
public bool Connect(  
    string phoneNumber,  
    string userName,  
    string userPassword  
);
```

Parameters

phoneNumber

A string which specifies the phone number to dial.

userName

A string which specifies the username which will be used to authenticate the session.

userPassword

A string which specifies the password which will be used to authenticate the session.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The **Connect** method establishes a dial-up networking connection with a service provider. A temporary phonebook entry will be created for the dial-up networking session, and this entry will be removed when the local host disconnects from the service provider.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [InternetDialer.Connect Overload List](#)

InternetDialer.Connect Method (String, String, String, String)

Establish a connection with a dial-up networking services provider.

[Visual Basic]

```
Overloads Public Function Connect( _  
    ByVal phoneNumber As String, _  
    ByVal userName As String, _  
    ByVal userPassword As String, _  
    ByVal userDomain As String _  
) As Boolean
```

[C#]

```
public bool Connect(  
    string phoneNumber,  
    string userName,  
    string userPassword,  
    string userDomain  
);
```

Parameters

phoneNumber

A string which specifies the phone number to dial.

userName

A string which specifies the username which will be used to authenticate the session.

userPassword

A string which specifies the password which will be used to authenticate the session.

userDomain

A string which specifies the domain on which the username and password will be authenticated. An empty string specifies the domain in which the Remote Access Server is a member.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The **Connect** method establishes a dial-up networking connection with a service provider. A temporary phonebook entry will be created for the dial-up networking session, and this entry will be removed when the local host disconnects from the service provider.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [InternetDialer.Connect Overload List](#)

InternetDialer.CreateEntry Method

Create a new entry in the current phonebook.

[Visual Basic]

```
Public Function CreateEntry() As Boolean
```

[C#]

```
public bool CreateEntry();
```

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The **CreateEntry** method displays a dialog box which allows the user to create a new phonebook entry on the system. If you do not wish to display a dialog box, use the **SaveEntry** method instead.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.DeleteEntry Method

Delete the current phonebook entry from the phonebook.

Overload List

Delete the current phonebook entry from the phonebook.

```
public bool DeleteEntry();
```

Delete a phonebook entry from the current phonebook.

```
public bool DeleteEntry(string);
```

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.DeleteEntry Method ()

Delete the current phonebook entry from the phonebook.

[Visual Basic]

Overloads Public Function DeleteEntry() As Boolean

[C#]

public bool DeleteEntry();

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The current phonebook entry name is specified by the **EntryName** property.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [InternetDialer.DeleteEntry Overload List](#)

InternetDialer.DeleteEntry Method (String)

Delete a phonebook entry from the current phonebook.

[Visual Basic]

```
Overloads Public Function DeleteEntry( _  
    ByVal entryName As String _  
) As Boolean
```

[C#]

```
public bool DeleteEntry(  
    string entryName  
);
```

Parameters

entryName

A string which specifies the phonebook entry name to delete.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [InternetDialer.DeleteEntry Overload List](#)

InternetDialer.Disconnect Method

Terminate the connection with the dial-up networking service provider.

[Visual Basic]

```
Public Function Disconnect() As Boolean
```

[C#]

```
public bool Disconnect();
```

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

This method may cause the current thread to block as the connection is terminated and the dial-up network device is being reset to its default state. Any active network connections using this dial-up networking connection will be terminated.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Dispose Method

Releases all resources used by [InternetDialer](#).

Overload List

Releases all resources used by [InternetDialer](#).

```
public void Dispose();
```

Releases the unmanaged resources allocated by the [InternetDialer](#) class and optionally releases the managed resources.

```
protected virtual void Dispose(bool);
```

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Dispose Method ()

Releases all resources used by [InternetDialer](#).

[Visual Basic]

```
NotOverridable Overloads Public Sub Dispose() _  
    Implements IDisposable.Dispose
```

[C#]

```
public void Dispose();
```

Implements

IDisposable.Dispose

Remarks

The **Dispose** method terminates any active connection and explicitly releases the resources allocated for this instance of the class. In some cases, better performance can be achieved if the programmer explicitly releases resources when they are no longer being used. The **Dispose** method provides explicit control over these resources.

Unlike the **Uninitialize** method, once the **Dispose** method has been called, that instance of the class cannot be re-initialized and you should not attempt to access class properties or invoke any methods. Note that this method can be called even if other references to the object are active.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [InternetDialer.Dispose Overload List](#)

InternetDialer.Dispose Method (Boolean)

Releases the unmanaged resources allocated by the [InternetDialer](#) class and optionally releases the managed resources.

[Visual Basic]

```
Overridable Overloads Protected Sub Dispose( _  
    ByVal disposing As Boolean _  
)
```

[C#]

```
protected virtual void Dispose(  
    bool disposing  
);
```

Parameters

disposing

A boolean value which should be specified as **true** to release both managed and unmanaged resources; **false** to release only unmanaged resources.

Remarks

The **Dispose** method terminates any active connection and explicitly releases the resources allocated for this instance of the class. In some cases, better performance can be achieved if the programmer explicitly releases resources when they are no longer being used. The **Dispose** method provides explicit control over these resources.

Unlike the **Uninitialize** method, once the **Dispose** method has been called, that instance of the class cannot be re-initialized and you should not attempt to access class properties or invoke any methods. Note that this method can be called even if other references to the object are active.

You should call **Dispose** in your derived class when you are finished using the derived class. The **Dispose** method leaves the derived class in an unusable state. After calling **Dispose**, you must release all references to the derived class and the **InternetDialer** class so the memory they were occupying can be reclaimed by garbage collection.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [InternetDialer.Dispose Overload List](#)

InternetDialer.EditEntry Method

Edit an existing phonebook entry in the current phonebook.

[Visual Basic]

```
Public Function EditEntry( _  
    ByVal entryName As String _  
) As Boolean
```

[C#]

```
public bool EditEntry(  
    string entryName  
);
```

Parameters

entryName

A string which specifies the name of the phonebook entry to be edited.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The **EditEntry** method edits the specified entry from the local phonebook. This will cause a dialog box to be displayed from which the user can change the connection information. If you do not want to display a dialog, then use the **SaveEntry** method instead.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Finalize Method

Destroys an instance of the class, releasing the resources allocated for the session and unloading the networking library.

[Visual Basic]

Overrides Protected Sub Finalize()

[C#]

protected override void Finalize();

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Initialize Method

Initialize an instance of the InternetDialer class.

Overload List

Initialize an instance of the InternetDialer class.

```
public bool Initialize();
```

Initialize an instance of the InternetDialer class.

```
public bool Initialize(string);
```

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [Uninitialize Method](#)

InternetDialer.Initialize Method ()

Initialize an instance of the InternetDialer class.

[Visual Basic]

Overloads Public Function Initialize() As Boolean

[C#]

public bool Initialize();

Return Value

A boolean value which specifies if the class was initialized successfully.

Remarks

The Initialize method can be used to explicitly initialize an instance of the InternetDialer class, loading the networking library and allocating resources for the current thread. Typically it is not necessary to explicitly call this method because the instance of the class is initialized by the class constructor. However, if the **Uninitialize** method is called, the class must be re-initialized before any other methods are called.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [InternetDialer.Initialize Overload List](#) | [Uninitialize Method](#)

InternetDialer.Initialize Method (String)

Initialize an instance of the InternetDialer class.

[Visual Basic]

```
Overloads Public Function Initialize( _  
    ByVal LicenseKey As String _  
) As Boolean
```

[C#]

```
public bool Initialize(  
    string LicenseKey  
);
```

Return Value

A boolean value which specifies if the class was initialized successfully.

Remarks

The Initialize method can be used to explicitly initialize an instance of the InternetDialer class, loading the networking library and allocating resources for the current thread. Typically an application would define the license key as a custom attribute, however this method can be used to initialize the class directly.

The runtime license key for your copy of SocketTools can be generated using the License Manager utility that is included with the product. Note that if you have installed an evaluation license, you will not have a runtime license key and cannot redistribute any applications which use the InternetDialer class.

Example

The following example shows how to use the Initialize method to initialize an instance of the class. This example assumes that the license key string has been defined in code.

```
SocketTools.InternetDialer xxxClient = new SocketTools.InternetDialer();  
  
if (xxxClient.Initialize(strLicenseKey) == false)  
{  
    MessageBox.Show(xxxClient.LastErrorString, "Error",  
        MessageBoxButtons.OK, MessageBoxIcon.Exclamation);  
    return;  
}  
  
Dim xxxClient As New SocketTools.InternetDialer  
  
If xxxClient.Initialize(strLicenseKey) = False Then  
    MsgBox(xxxClient.LastErrorString, vbIconExclamation)  
    Exit Sub  
End If
```

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [InternetDialer.Initialize Overload List](#) | [RuntimeLicenseAttribute Class](#) | [Uninitialize Method](#)

InternetDialer.LoadEntry Method

Reload the current phonebook entry from the phonebook.

Overload List

Reload the current phonebook entry from the phonebook.

```
public bool LoadEntry();
```

Load the specified entry from the current phonebook.

```
public bool LoadEntry(string);
```

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.LoadEntry Method ()

Reload the current phonebook entry from the phonebook.

[Visual Basic]

Overloads Public Function LoadEntry() As Boolean

[C#]

public bool LoadEntry();

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The current entry from the phonebook will be reloaded and any changes made to the current entry will be abandoned.

The current phonebook entry name is specified by the **EntryName** property.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [InternetDialer.LoadEntry Overload List](#)

InternetDialer.LoadEntry Method (String)

Load the specified entry from the current phonebook.

[Visual Basic]

```
Overloads Public Function LoadEntry( _  
    ByVal entryName As String _  
) As Boolean
```

[C#]

```
public bool LoadEntry(  
    string entryName  
);
```

Parameters

entryName

A string which specifies the name of a phonebook entry.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The **LoadEntry** method loads the specified entry from the current phonebook and sets the class properties to match the configuration.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [InternetDialer.LoadEntry Overload List](#)

InternetDialer.RenameEntry Method

Rename an existing phonebook entry.

[Visual Basic]

```
Public Function RenameEntry( _  
    ByVal oldEntryName As String, _  
    ByVal newEntryName As String _  
) As Boolean
```

[C#]

```
public bool RenameEntry(  
    string oldEntryName,  
    string newEntryName  
);
```

Parameters

oldEntryName

A string which specifies the phonebook entry which will be renamed.

newEntryName

A string which specifies the new name for the phonebook entry.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The **RenameEntry** method renames the specified entry in the local phonebook. The new entry name must not already exist in the phonebook.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.Reset Method

Reset the internal state of the object, resetting all properties to their default values.

[Visual Basic]

```
Public Sub Reset()
```

[C#]

```
public void Reset();
```

Remarks

The **Reset** method returns the object to its default state. If a socket has been allocated, it will be released and any active connections will be terminated. All properties will be reset to their default values.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.SaveEntry Method

Save the current phonebook entry settings.

Overload List

Save the current phonebook entry settings.

```
public bool SaveEntry();
```

Save the current settings to the specified phonebook entry in the current phonebook.

```
public bool SaveEntry(string);
```

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.SaveEntry Method ()

Save the current phonebook entry settings.

[Visual Basic]

Overloads Public Function SaveEntry() As Boolean

[C#]

public bool SaveEntry();

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The **SaveEntry** method saves the current phonebook entry settings, based on the class property values. If the entry does not exist, it will be created. If the entry does exist, it will be overwritten. Note that unlike the **CreateEntry** method, this method does not display any user-interface dialogs.

The current phonebook entry name is specified by the **EntryName** property.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [InternetDialer.SaveEntry Overload List](#)

InternetDialer.SaveEntry Method (String)

Save the current settings to the specified phonebook entry in the current phonebook.

[Visual Basic]

```
Overloads Public Function SaveEntry( _  
    ByVal entryName As String _  
) As Boolean
```

[C#]

```
public bool SaveEntry(  
    string entryName  
);
```

Parameters

entryName

A string which specifies the name of the phonebook entry.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The **SaveEntry** method saves the specified entry to the current phonebook, based on the class property values. If the entry does not exist, it will be created. If the entry does exist, it will be overwritten. Note that unlike the **CreateEntry** method, this method does not display any user-interface dialogs.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [InternetDialer.SaveEntry Overload List](#)

InternetDialer.Uninitialize Method

Uninitialize the class library and release any resources allocated for the current thread.

[Visual Basic]

```
Public Sub Uninitialize()
```

[C#]

```
public void Uninitialize();
```

Remarks

The **Uninitialize** method terminates any active connection, releases resources allocated for the current thread and unloads the networking library. After this method has been called, no further client operations may be performed until the class instance has been re-initialized.

If the **Initialize** method is explicitly called by the application, it should be matched by a call to the **Uninitialize** method when that instance of the class is no longer needed.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#) | [Initialize Method](#)

InternetDialer Events

The events of the **InternetDialer** class are listed below. For a complete list of **InternetDialer** class members, see the [InternetDialer Members](#) topic.

Public Instance Events

OnCancel	Occurs when a blocking client operation is canceled.
OnConnect	Occurs when a connection is established with the service provider.
OnDisconnect	Occurs when the dial-up networking connection is terminated.
OnError	Occurs when an client operation fails.
OnStatus	Occurs when the when the connection state changes.
OnTimeout	Occurs when a blocking operation fails to complete before the timeout period elapses.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.OnCancel Event

Occurs when a blocking client operation is canceled.

[Visual Basic]

```
Public Event OnCancel As EventHandler
```

[C#]

```
public event EventHandler OnCancel;
```

Remarks

The **OnCancel** event is generated when a blocking client operation, such as sending or receiving data, is canceled with the **Cancel** method. To assist in determining which operation was canceled, check the value of the **Status** property.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.OnConnect Event

Occurs when a connection is established with the service provider.

[Visual Basic]

Public Event OnConnect As EventHandler

[C#]

public event EventHandler OnConnect;

Remarks

The **OnConnect** event occurs when a connection is made with a service provider as a result of a **Connect** method call. When the **Connect** method is called and the **Blocking** property is set to **false**, a dial-up networking connection is initiated, but the connection is not actually established until after this event occurs. Between the time connection process is started and this event fires, no operation may be performed by the client other than calling the **Disconnect** method.

This event is only generated if the client is in non-blocking mode.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.OnDisconnect Event

Occurs when the dial-up networking connection is terminated.

[Visual Basic]

```
Public Event OnDisconnect As EventHandler
```

[C#]

```
public event EventHandler OnDisconnect;
```

Remarks

The **OnDisconnect** event occurs when the dial-up networking connection has been terminated. This event is only generated if the client is in non-blocking mode.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.OnError Event

Occurs when an client operation fails.

```
[Visual Basic]
Public Event OnError As OnErrorHandler
```

```
[C#]
public event OnErrorHandler OnError;
```

Event Data

The event handler receives an argument of type [InternetDialer.ErrorEventArgs](#) containing data related to this event. The following **InternetDialer.ErrorEventArgs** properties provide information specific to this event.

Property	Description
Description	Gets a value which describes the last error that has occurred.
Error	Gets a value which specifies the last error that has occurred.

Remarks

The **OnError** event occurs when a client operation fails.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.ErrorEventArgs Class

Provides data for the OnError event.

For a list of all members of this type, see [InternetDialer.ErrorEventArgs Members](#).

System.Object

System.EventArgs

SocketTools.InternetDialer.ErrorEventArgs

[Visual Basic]

Public Class InternetDialer.ErrorEventArgs

Inherits EventArgs

[C#]

public class InternetDialer.ErrorEventArgs : EventArgs

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

ErrorEventArgs specifies the numeric error code and a description of the error that has occurred.

An [OnError](#) event occurs when a method fails.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[InternetDialer.ErrorEventArgs Members](#) | [SocketTools Namespace](#)

InternetDialer.ErrorEventArgs Members

[InternetDialer.ErrorEventArgs overview](#)

Public Instance Constructors

 InternetDialer.ErrorEventArgs Constructor	Initializes a new instance of the InternetDialer.ErrorEventArgs class.

Public Instance Properties

Description	Gets a value which describes the last error that has occurred.
Error	Gets a value which specifies the last error that has occurred.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetDialer.ErrorEventArgs Class](#) | [SocketTools Namespace](#)

InternetDialer.ErrorEventArgs Constructor

Initializes a new instance of the [InternetDialer.ErrorEventArgs](#) class.

[Visual Basic]

```
Public Sub New()
```

[C#]

```
public InternetDialer.ErrorEventArgs();
```

See Also

[InternetDialer.ErrorEventArgs Class](#) | [SocketTools Namespace](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InternetDialer.ErrorEventArgs Properties

The properties of the **InternetDialer.ErrorEventArgs** class are listed below. For a complete list of **InternetDialer.ErrorEventArgs** class members, see the [InternetDialer.ErrorEventArgs Members](#) topic.

Public Instance Properties

Description	Gets a value which describes the last error that has occurred.
Error	Gets a value which specifies the last error that has occurred.

See Also

[InternetDialer.ErrorEventArgs Class](#) | [SocketTools Namespace](#)

InternetDialer.ErrorEventArgs.Description Property

Gets a value which describes the last error that has occurred.

[Visual Basic]

Public ReadOnly Property Description As String

[C#]

public string Description {get;}

Property Value

A string which describes the last error that has occurred.

See Also

[InternetDialer.ErrorEventArgs Class](#) | [SocketTools Namespace](#) | [Error Property](#)

InternetDialer.ErrorEventArgs.Error Property

Gets a value which specifies the last error that has occurred.

[Visual Basic]

Public **ReadOnly** **Property** **Error** As [ErrorCode](#)

[C#]

public [InternetDialer.ErrorCode](#) **Error** {get;}

Property Value

[ErrorCode](#) enumeration which specifies the error.

See Also

[InternetDialer.ErrorEventArgs Class](#) | [SocketTools Namespace](#) | [Description Property](#)

InternetDialer.OnStatus Event

Occurs when the when the connection state changes.

```
[Visual Basic]
Public Event OnStatus As OnStatusEventHandler
```

```
[C#]
public event OnStatusEventHandler OnStatus;
```

Event Data

The event handler receives an argument of type [InternetDialer.StatusEventArgs](#) containing data related to this event. The following **InternetDialer.StatusEventArgs** properties provide information specific to this event.

Property	Description
Description	Gets a description of the current dial-up networking connection status.
Status	Gets a value which specifies the current status of the dial-up networking connection.

Remarks

The **OnStatus** event is generated when the status of the connection changes. Typically this occurs when a connection is being established with a dial-up networking server.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.StatusEventArgs Class

Provides data for the OnStatus event.

For a list of all members of this type, see [InternetDialer.StatusEventArgs Members](#).

System.Object

System.EventArgs

SocketTools.InternetDialer.StatusEventArgs

[Visual Basic]

Public Class InternetDialer.StatusEventArgs

Inherits EventArgs

[C#]

public class InternetDialer.StatusEventArgs : EventArgs

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

StatusEventArgs specifies the status code and a description of the status for the last status change that has occurred.

The [OnStatus](#) event occurs whenever the status of the dial-up networking connection changes.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[InternetDialer.StatusEventArgs Members](#) | [SocketTools Namespace](#)

InternetDialer.StatusEventArgs Members

[InternetDialer.StatusEventArgs overview](#)

Public Instance Constructors

 InternetDialer.StatusEventArgs Constructor	Initializes a new instance of the InternetDialer.StatusEventArgs class.

Public Instance Properties

Description	Gets a description of the current dial-up networking connection status.
Status	Gets a value which specifies the current status of the dial-up networking connection.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetDialer.StatusEventArgs Class](#) | [SocketTools Namespace](#)

InternetDialer.StatusEventArgs Constructor

Initializes a new instance of the [InternetDialer.StatusEventArgs](#) class.

[Visual Basic]

```
Public Sub New()
```

[C#]

```
public InternetDialer.StatusEventArgs();
```

See Also

[InternetDialer.StatusEventArgs Class](#) | [SocketTools Namespace](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InternetDialer.StatusEventArgs Properties

The properties of the **InternetDialer.StatusEventArgs** class are listed below. For a complete list of **InternetDialer.StatusEventArgs** class members, see the [InternetDialer.StatusEventArgs Members](#) topic.

Public Instance Properties

Description	Gets a description of the current dial-up networking connection status.
Status	Gets a value which specifies the current status of the dial-up networking connection.

See Also

[InternetDialer.StatusEventArgs Class](#) | [SocketTools Namespace](#)

InternetDialer.StatusEventArgs.Description Property

Gets a description of the current dial-up networking connection status.

[Visual Basic]

Public ReadOnly Property Description As String

[C#]

public string Description {get;}

Property Value

A string which describes the connection status.

See Also

[InternetDialer.StatusEventArgs Class](#) | [SocketTools Namespace](#) | [Error Property](#)

InternetDialer.StatusEventArgs.Status Property

Gets a value which specifies the current status of the dial-up networking connection.

[Visual Basic]

Public **ReadOnly** **Property** **Status** **As** [DialerStatus](#)

[C#]

public [InternetDialer.DialerStatus](#) **Status** {get;}

Property Value

A [DialerStatus](#) enumeration value which specifies the current status.

See Also

[InternetDialer.StatusEventArgs Class](#) | [SocketTools Namespace](#)

InternetDialer.OnTimeout Event

Occurs when a blocking operation fails to complete before the timeout period elapses.

[Visual Basic]

Public Event OnTimeout As EventHandler

[C#]

public event EventHandler OnTimeout;

Remarks

The **OnTimeout** event occurs when a blocking operation, such as sending or receiving data on the client, fails to complete before the specified timeout period elapses. The timeout period for a blocking operation can be adjusted by setting the **Timeout** property.

This event is only generated if the client is in blocking mode.

See Also

[InternetDialer Class](#) | [SocketTools Namespace](#)

InternetDialer.OnErrorHandler Delegate

Represents the method that will handle the [OnError](#) event.

[Visual Basic]

```
Public Delegate Sub InternetDialer.OnErrorHandler( _  
    ByVal sender As Object, _  
    ByVal e As EventArgs _  
)
```

[C#]

```
public delegate void InternetDialer.OnErrorHandler(  
    object sender,  
    EventArgs e  
);
```

Parameters

sender

The source of the event.

e

An [EventArgs](#) that contains the event data.

Remarks

When you create an **OnErrorEventHandler** delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, until you remove the delegate.

Note that the declaration of your event handler must have the same parameters as the **OnErrorEventHandler** delegate declaration.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[SocketTools Namespace](#)

InternetDialer.OnStatusEventHandler Delegate

Represents the method that will handle the [OnStatus](#) event.

[Visual Basic]

```
Public Delegate Sub InternetDialer.OnStatusEventHandler( _  
    ByVal sender As Object, _  
    ByVal e As StatusEventArgs _  
)
```

[C#]

```
public delegate void InternetDialer.OnStatusEventHandler(  
    object sender,  
    StatusEventArgs e  
);
```

Parameters

sender

The source of the event.

e

A [StatusEventArgs](#) object that contains the event data.

Remarks

When you create an **OnStatusEventHandler** delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, until you remove the delegate.

Note that the declaration of your event handler must have the same parameters as the **OnStatusEventHandler** delegate declaration.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[SocketTools Namespace](#)

InternetDialer.DialerStatus Enumeration

The current state of the connection.

[Visual Basic]

Public Enum InternetDialer.DialerStatus

[C#]

public enum InternetDialer.DialerStatus

Members

Member Name	Description
Unused	No connection has been established.
OpenPort	The communications port is about to be opened.
PortOpened	The communications port has been opened.
ConnectDevice	A device is about to be connected.
DeviceConnected	A device has been connected successfully.
AllDevicesConnected	All devices have been connected.
Authenticate	Authenticating username and password.
AuthNotify	An authentication event has occurred.
AuthRetry	Requesting authentication with new credentials.
AuthCallback	The remote server has requested a callback number.
AuthChangePassword	The user has requested to change the password.
AuthProject	Registering computer on the network.
AuthLinkSpeed	The link speed calculation phase is starting.
AuthAck	An authentication request is being acknowledged.
ReAuthenticate	Authenticating username and password.
Authenticated	The user has been authenticated.
PrepareForCallback	The line is about to be disconnected in preparation for callback.
WaitForModemReset	The modem is resetting itself in preparation for callback.
WaitForCallback	Waiting for callback from remote server.
Projected	Protocol specific information has been negotiated.
StartAuthentication	User authentication is being initiated.
CallbackComplete	Callback completed and resuming authentication.
LogonNetwork	Logging on to the network.
SubEntryConnected	A subentry has been connected.
SubEntryDisconnected	A subentry has been disconnected.

Interactive	Initiating interactive login session.
RetryAuthentication	Retrying user authentication.
CallbackSetByCaller	Callback has been set by caller.
PasswordExpired	Password has expired.
Connected	Connected to remote server.
Disconnected	Disconnected from remote server.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[SocketTools Namespace](#)

InternetDialer.ErrorCode Enumeration

The error codes used by the class.

[Visual Basic]

```
Public Enum InternetDialer.ErrorCode
```

[C#]

```
public enum InternetDialer.ErrorCode
```

Remarks

The **ErrorCode** enumeration specifies what error has occurred when a method fails. The current error code may be determined by checking the value of the **LastError** property.

Note that the last error code is only meaningful if the previous operation has failed.

Members

Member Name	Description
None	No error.
OperationCanceled	The blocking operation has been canceled.
InvalidDevice	The specified file type is invalid or not a regular file.
TooManyParameters	The maximum number of function parameters has been exceeded.
DeviceNotFound	The specified device could not be found.
OperationTimeout	The specified operation has timed out.
Pending	An operation is pending.
InvalidPortHandle	An invalid port handle was detected.
PortAlreadyOpen	The specified port is already open.
BufferTooSmall	The caller's buffer is too small.
WrongInfoSpecified	Incorrect information was specified.
CannotSetPortInfo	The port information cannot be set.
PortNotConnected	The specified port is not connected.
EventInvalid	An invalid event was detected.
DeviceDoesNotExist	A device was specified that does not exist.
DevicetypeDoesNotExist	A device type was specified that does not exist.
BufferInvalid	An invalid buffer was specified.
RouteNotAvailable	A route was specified that is not available.
RouteNotAllocated	A route was specified that is not allocated.
InvalidCompressionSpecified	An invalid compression was specified.
OutOfBuffers	There were insufficient buffers available.
PortNotFound	The specified port was not found.

AsyncRequestPending	An asynchronous request is pending.
AlreadyDisconnecting	The modem or other connecting device is already disconnecting.
PortNotOpen	The specified port is not open.
PortDisconnected	A connection to the remote computer could not be established.
NoEndpoints	No endpoints could be determined.
CannotOpenPhonebook	The system could not open the phone book file.
CannotLoadPhonebook	The system could not load the phone book file.
CannotFindPhonebookEntry	The system could not find the phone book entry for this connection.
CannotWritePhonebook	The system could not update the phone book file.
CorruptPhonebook	The system found invalid information in the phone book file.
CannotLoadString	A string could not be loaded.
KeyNotFound	A key could not be found.
Disconnection	The connection was terminated by the remote computer before it could be completed.
RemoteDisconnection	The connection was closed by the remote computer.
HardwareFailure	The modem or other connecting device was disconnected due to hardware failure.
UserDisconnection	The user disconnected the modem or other connecting device.
InvalidSize	An incorrect structure size was detected.
PortNotAvailable	The modem or other connecting device is already in use or is not configured properly.
CannotProjectClient	Your computer could not be registered on the remote network.
Unknown	There was an unknown error.
WrongDeviceAttached	The device attached to the port is not the one expected.
BadString	A string was detected that could not be converted.
RequestTimeout	The request has timed out.
CannotGetLana	No asynchronous net is available.
NetBIOSError	An error has occurred involving NetBIOS.
ServerOutOfResources	The server cannot allocate NetBIOS resources needed to support the client.
NameExistsOnNet	One of your computer's NetBIOS names is already registered on the remote network.
ServerGeneralNetFailure	A network adapter at the server failed.

MsgAliasNotAdded	You will not receive network message popups.
AuthInternal	There was an internal authentication error.
RestrictedLogonHours	The account is not permitted to log on at this time of day.
AcctDisabled	The account is disabled.
PasswdExpired	The password for this account has expired.
NoDialInPermission	The account does not have permission to dial in.
ServerNotResponding	The remote access server is not responding.
FromDevice	The modem or other connecting device has reported an error.
UnrecognizedResponse	There was an unrecognized response from the modem or other connecting device.
MacroNotFound	A macro required by the modem or other connecting device was not found in the configuration file.
MacroNotDefined	A command or response in the configuration file refers to an undefined macro.
MessageMacroNotFound	The message macro was not found in the configuration file.
DefaultOffMacroNotFound	The configuration file contains an undefined macro.
FileCouldNotBeOpened	The configuration file could not be opened.
DevicenameTooLong	The device name in the configuration file is too long.
DevicenameNotFound	The configuration file refers to an unknown device name.
NoResponses	The configuration file contains no responses for the command.
NoCommandFound	The configuration file is missing a command.
WrongKeySpecified	There was an attempt to set a macro not listed in configuration file.
UnknownDeviceType	The configuration file refers to an unknown device type.
AllocatingMemory	The system has run out of memory.
PortNotConfigured	The modem or other connecting device is not properly configured.
DeviceNotReady	The modem or other connecting device is not functioning.
ReadingIniFile	The system was unable to read the configuration file.
NoConnection	The connection was terminated.
BadUsageInIniFile	The usage parameter in the configuration file is invalid.

ReadingSectionname	The system was unable to read the section name from the configuration file.
ReadingDeviceType	The system was unable to read the device type from the configuration file.
ReadingDeviceName	The system was unable to read the device name from the configuration file.
ReadingUsage	The system was unable to read the usage from the configuration file.
ReadingMaxConnectBps	The system was unable to read the maximum connection BPS rate from the configuration file.
ReadingMaxCarrierBps	The system was unable to read the maximum carrier connection speed from the configuration file.
LineBusy	The phone line is busy.
VoiceAnswer	A person answered instead of a modem or other connecting device.
NoAnswer	The remote computer did not respond.
NoCarrier	The system could not detect the carrier.
NoDialtone	There was no dial tone.
InCommand	The modem or other connecting device reported a general error.
WritingSectionname	There was an error in writing the section name.
WritingDeviceType	There was an error in writing the device type.
WritingDeviceName	There was an error in writing the device name.
WritingMaxConnectBps	There was an error in writing the maximum connection speed..
WritingMaxCarrierBps	There was an error in writing the maximum carrier speed.
WritingUsage	There was an error in writing the usage.
WritingDefaultOff	There was an error in writing the default-off.
ReadingDefaultOff	There was an error in reading the default-off.
EmptyIniFile	The configuration file is empty.
AuthenticationFailure	Access was denied because the username and/or password was invalid on the domain.
PortOrDevice	There was a hardware failure in the modem or other connecting device.
NotBinaryMacro	An internal error has occurred.
DcbNotFound	An internal error has occurred.
StateMachinesNotStarted	The state machines are not started.
StateMachinesAlreadyStarted	The state machines are already started.

PartialResponseLooping	The response looping did not complete.
UnknownResponseKey	A response keyname in the configuration file is not in the expected format.
RecvBufFull	The modem or other connecting device response caused a buffer overflow.
CmdTooLong	The expanded command in the configuration file is too long.
UnsupportedBps	The modem moved to a connection speed not supported by the COM driver.
UnexpectedResponse	Device response received when none expected.
InteractiveMode	The connection needs information from you, but the application does not allow user interaction.
BadCallbackNumber	The callback number is invalid.
InvalidAuthState	The authorization state is invalid.
WritingInitBps	An internal error has occurred.
X25Diagnostic	There was an error related to the X.25 protocol.
AcctExpired	The account has expired.
ChangingPassword	There was an error changing the password on the domain.
Overrun	Serial overrun errors were detected while communicating with the modem.
RasmanCannotInitialize	A configuration error on this computer is preventing this connection.
BiplexPortNotAvailable	The two-way port is initializing, wait a few seconds and redial.
NoActiveIsdnLines	No active ISDN lines are available.
NoIsdnChannelsAvailable	No ISDN channels are available to make the call.
TooManyLineErrors	Too many errors occurred because of poor phone line quality.
IpConfiguration	The Remote Access Service IP configuration is unusable.
NoIpAddresses	No IP addresses are available in the static pool of Remote Access Service IP addresses.
PppTimeout	The connection was terminated because the remote computer did not respond in a timely manner.
PppRemoteTerminated	The connection was terminated by the remote computer.
PppNoProtocolsConfigured	A connection to the remote computer could not be established.
PppNoResponse	The remote computer did not respond.

PppInvalidPacket	Invalid data was received from the remote computer.
PhoneNumberTooLong	The phone number, including prefix and suffix, is too long.
IpxcpNoDialoutConfigured	The IPX protocol cannot dial out on the modem because this computer is not configured for dialing out.
IpxcpNoDialinConfigured	The IPX protocol cannot dial in on the modem because this computer is not configured for dialing in.
IpxcpDialoutAlreadyActive	The IPX protocol cannot be used for dialing out on more than one modem.
AccessingTcpcfgDll	Cannot access TCPCFG.DLL.
NoIpRasAdapter	The system cannot find an IP adapter.
SlipRequiresIp	SLIP cannot be used unless the IP protocol is installed.
ProjectionNotComplete	Computer registration is not complete.
ProtocolNotConfigured	The protocol is not configured.
PppNotConverging	Your computer and the remote computer could not agree on PPP control protocols.
PppCpRejected	A connection to the remote computer could not be completed.
PppLcpTerminated	The PPP link control protocol was terminated.
PppRequiredAddressRejected	The requested address was rejected by the server.
PppNcpTerminated	The remote computer terminated the control protocol.
PppLoopbackDetected	Loopback was detected.
PppNoAddressAssigned	The server did not assign an address.
CannotUseLogonCredentials	The authentication protocol required by the remote server cannot use the stored password.
TapiConfiguration	An invalid dialing rule was detected.
NoLocalEncryption	The local computer does not support the required data encryption type.
NoRemoteEncryption	The remote computer does not support the required data encryption type.
RemoteRequiresEncryption	The remote computer requires data encryption.
IpxcpNetNumberConflict	The system cannot use the IPX network number assigned by the remote computer.
InvalidSMM	An internal error has occurred.
SMMUninitialized	An internal error has occurred.

NoMacForPort	An internal error has occurred.
SmmTimeout	An internal error has occurred.
BadPhoneNumber	An invalid telephone number has been specified.
WrongModule	An internal error has occurred.
InvalidCallbackNumber	The callback number contains an invalid character.
ScriptSyntax	A syntax error was encountered while processing a script.
HangupFailed	The connection could not be disconnected because it was created by the multi-protocol router.
BundleNotFound	The system could not find the multi-link bundle.
CannotDoCustomdial	The system cannot perform automated dial because this connection has a custom dialer specified.
DialAlreadyInProgress	This connection is already being dialed.
RasautoCannotInitialize	Remote Access Services could not be started automatically.
ConnectionAlreadyShared	Internet Connection Sharing is already enabled on the connection.
SharingChangeFailed	An error occurred while the existing Internet Connection Sharing settings were being changed.
SharingRouterInstall	An error occurred while routing capabilities were being enabled.
ShareConnectionFailed	An error occurred while Internet Connection Sharing was being enabled for the connection.
SharingPrivateInstall	An error occurred while the local network was being configured for sharing.
CannotShareConnection	Internet Connection Sharing cannot be enabled.
NoSmartCardReader	No smart card reader is installed.
SharingAddressExists	Internet Connection Sharing cannot be enabled.
NoCertificate	A certificate could not be found.
SharingMultipleAddresses	Internet Connection Sharing cannot be enabled.
FailedToEncrypt	The connection attempt failed because of failure to encrypt data.
BadAddressSpecified	The specified destination is not reachable.
ConnectionReject	The remote computer rejected the connection attempt.
Congestion	The connection attempt failed because the network is busy.
Incompatible	The remote computer's network hardware is incompatible with the type of call requested.
Numberchanged	The connection attempt failed because the

	destination number has changed.
Tempfailure	The connection attempt failed because of a temporary failure.
Blocked	The call was blocked by the remote computer.
Donotdisturb	The call could not be connected because the remote computer has invoked the Do Not Disturb feature.
Outoforder	The connection attempt failed because the modem on the remote computer is out of order.
UnableToAuthenticateServer	It was not possible to verify the identity of the server.
SmartCardRequired	To dial out using this connection you must use a smart card.
InvalidFunctionForEntry	An attempted function is not valid for this connection.
CertForEncryptionNotFound	The connection requires a certificate, and no valid certificate was found.
SharingRasConflict	Network Address Translation must be removed before enabling Internet Connection Sharing.
SharingNoPrivateLan	Internet Connection Sharing cannot be enabled.
NoDiffUserAtLogon	You cannot dial using this connection at logon time.
NoRegCertAtLogon	You cannot dial using this connection at logon time.
OakleyNoCert	The L2TP connection attempt failed because there is no valid machine certificate on your computer for security authentication.
OakleyAuthFail	The L2TP connection attempt failed because the security layer could not authenticate the remote computer.
OakleyAttribFail	The L2TP connection attempt failed because the security layer could not negotiate compatible parameters with the remote computer.
OakleyGeneralProcessing	The L2TP connection attempt failed because the security layer encountered a processing error during initial negotiations with the remote computer.
OakleyNoPeerCert	The L2TP connection attempt failed because certificate validation on the remote computer failed.
OakleyNoPolicy	The L2TP connection attempt failed because security policy for the connection was not found.
OakleyTimedOut	The L2TP connection attempt failed because security negotiation timed out.
OakleyError	The L2TP connection attempt failed because an error occurred while negotiating security.
UnknownFramedProtocol	The Framed Protocol RADIUS attribute for this user is not PPP.

WrongTunnelType	The Tunnel Type RADIUS attribute for this user is not correct.
UnknownServiceType	The Service Type RADIUS attribute for this user is neither Framed nor Callback Framed.
ConnectingDeviceNotFound	A connection to the remote computer could not be established because the modem was not found or was busy.
NoEapTlsCertificate	A certificate could not be found that can be used with this Extensible Authentication Protocol.
SharingHostAddressConflict	Internet Connection Sharing cannot be enabled.
AutomaticVpnFailed	Unable to establish the VPN connection.
ValidatingServerCert	Unable to verify the digital certificate sent by the server.
ReadSCard	The card supplied was not recognized, please check that the card is inserted correctly, and fits tightly
InvalidPEAPCookieConfig	The PEAP configuration stored in the session cookie does not match the current session configuration
InvalidPEAPCookieUser	The PEAP identity stored in the session cookie does not match the current identity
InvalidMSCHAPV2Config	You cannot dial using this connection at logon time, because it is configured to use logged on user's credentials
InvalidLicense	The license for this product is invalid.
ProductNotLicensed	This product is not licensed to perform this operation.
NotImplemented	This function has not been implemented on this platform.
OperationNotSupported	The specified operation is not supported.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[SocketTools Namespace](#) | [LastError Property](#) | [LastErrorString Property](#) | [OnError Event](#)

InternetDialer.RasFramingProtocol Enumeration

The supported framing protocols.

This enumeration has a FlagsAttribute attribute that allows a bitwise combination of its member values.

```
[Visual Basic]
<Flags>
Public Enum InternetDialer.RasFramingProtocol
```

```
[C#]
[Flags]
public enum InternetDialer.RasFramingProtocol
```

Members

Member Name	Description	Value
Ppp	Point-to-Point Protocol (PPP). This is the most common protocol used by Internet Service Providers (ISPs).	1
Slip	Serial Line Internet Protocol (SLIP). This is a protocol commonly used when connecting to older UNIX systems.	2
Ras	A proprietary Microsoft protocol implemented in Windows.	4

Requirements

- Namespace: [SocketTools](#)
- Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[SocketTools Namespace](#)

InternetDialer.RasNetworkProtocol Enumeration

The supported networking protocols.

This enumeration has a FlagsAttribute attribute that allows a bitwise combination of its member values.

[Visual Basic]

<Flags>

Public Enum InternetDialer.RasNetworkProtocol

[C#]

[Flags]

public enum InternetDialer.RasNetworkProtocol

Remarks

These values may be combined if multiple protocols should be negotiated when the connection is established.

Note that unless there is a specific need for the application to use the NetBEUI or IPX protocols, it is recommended that only the TCP/IP protocol be specified.

Members

Member Name	Description	Value
NetBeui	Negotiate the NetBEUI protocol.	1
Ipx	Negotiate the IPX protocol.	2
TcpIp	Negotiate the TCP/IP protocol.	4

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[SocketTools Namespace](#)

InternetDialer.RasTerminalMode Enumeration

The supported interactive terminal modes

This enumeration has a FlagsAttribute attribute that allows a bitwise combination of its member values.

```
[Visual Basic]
<Flags>
Public Enum InternetDialer.RasTerminalMode
```

```
[C#]
[Flags]
public enum InternetDialer.RasTerminalMode
```

Remarks

These values may be combined if multiple terminal modes should be used when the connection is established. If scripting has been enabled by setting the **ScriptFile** property, no terminal window should be displayed after the connection. This is because scripting has its own terminal implementation.

Members

Member Name	Description	Value
None	No terminal window is displayed.	0
BeforeDial	Terminal window is displayed before dialing.	1
AfterDial	Terminal window is displayed after dialing. Do not use if scripting has been enabled.	2

Requirements

- Namespace:** [SocketTools](#)
- Assembly:** SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[SocketTools Namespace](#)

InternetDialer.RuntimeLicenseAttribute Class

Attribute that defines the runtime license key for the class.

For a list of all members of this type, see [InternetDialer.RuntimeLicenseAttribute Members](#).

System.Object

System.Attribute

SocketTools.InternetDialer.RuntimeLicenseAttribute

[Visual Basic]

```
<AttributeUsage(ValidOn:=AttributeTargets.Assembly, AllowMultiple:=False,
    Inherited:=True)>
```

```
Public Class InternetDialer.RuntimeLicenseAttribute
    Inherits Attribute
```

[C#]

```
[AttributeUsage(ValidOn=AttributeTargets.Assembly, AllowMultiple=False,
    Inherited=True)]
```

```
public class InternetDialer.RuntimeLicenseAttribute : Attribute
```

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

The RuntimeLicense attribute is used to define the runtime license key that will be used when an instance of the class is created. This attribute is defined in the assembly information module for the language, such as AssemblyInfo.cs when programming C#. The runtime license key must be defined if you wish to redistribute your application.

The runtime license key for your copy of SocketTools can be generated using the License Manager utility that is included with the product. Note that if you have installed an evaluation license, you will not have a runtime license key and cannot redistribute any applications which use the InternetDialer class.

Example

```
<Assembly: SocketTools.InternetDialer.RuntimeLicense("abcdefghijklmnop")>
```

```
[assembly: SocketTools.InternetDialer.RuntimeLicense("abcdefghijklmnop")]
```

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[InternetDialer.RuntimeLicenseAttribute Members](#) | [SocketTools Namespace](#)

InternetDialer.RuntimeLicenseAttribute Members

[InternetDialer.RuntimeLicenseAttribute overview](#)

Public Instance Constructors

 InternetDialer.RuntimeLicenseAttribute Constructor	Constructor for the RuntimeLicense attribute which defines the runtime license key.

Public Instance Properties

LicenseKey	Returns the value of the runtime license key.
TypeId (inherited from Attribute)	When implemented in a derived class, gets a unique identifier for this Attribute.

Public Instance Methods

Equals (inherited from Attribute)	
GetHashCode (inherited from Attribute)	Returns the hash code for this instance.
GetType (inherited from Object)	Gets the Type of the current instance.
IsDefaultAttribute (inherited from Attribute)	When overridden in a derived class, returns an indication whether the value of this instance is the default value for the derived class.
Match (inherited from Attribute)	When overridden in a derived class, returns a value indicating whether this instance equals a specified object.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetDialer.RuntimeLicenseAttribute Class](#) | [SocketTools Namespace](#)

InternetDialer.RuntimeLicenseAttribute Constructor

Constructor for the RuntimeLicense attribute which defines the runtime license key.

[Visual Basic]

```
Public Sub New( _  
    ByVal LicenseKey As String _  
)
```

[C#]

```
public InternetDialer.RuntimeLicenseAttribute(  
    string LicenseKey  
);
```

Parameters

licenseKey

A string argument which specifies the runtime license key which will be used to initialize the class library.

Remarks

The runtime license key for your copy of SocketTools can be generated using the License Manager utility that is included with the product. Note that if you have installed an evaluation license, you will not have a runtime license key and cannot redistribute any applications which use the InternetDialer class.

See Also

[InternetDialer.RuntimeLicenseAttribute Class](#) | [SocketTools Namespace](#)

InternetDialer.RuntimeLicenseAttribute Properties

The properties of the **InternetDialer.RuntimeLicenseAttribute** class are listed below. For a complete list of **InternetDialer.RuntimeLicenseAttribute** class members, see the [InternetDialer.RuntimeLicenseAttribute Members](#) topic.

Public Instance Properties

LicenseKey	Returns the value of the runtime license key.
TypeId (inherited from Attribute)	When implemented in a derived class, gets a unique identifier for this Attribute.

See Also

[InternetDialer.RuntimeLicenseAttribute Class](#) | [SocketTools Namespace](#)

InternetDialer.RuntimeLicenseAttribute.LicenseKey Property

Returns the value of the runtime license key.

[Visual Basic]

Public Property LicenseKey As String

[C#]

public string LicenseKey {get; set;}

Property Value

A string which contains the runtime license key.

See Also

[InternetDialer.RuntimeLicenseAttribute Class](#) | [SocketTools Namespace](#)

InternetDialerException Class

The exception that is thrown when a client error occurs.

For a list of all members of this type, see [InternetDialerException Members](#).

System.Object

System.Exception

System.ApplicationException

SocketTools.InternetDialerException

[Visual Basic]

Public Class InternetDialerException

Inherits ApplicationException

[C#]

public class InternetDialerException : ApplicationException

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

A InternetDialerException is thrown by the InternetDialer class when an error occurs.

The default constructor for the InternetDialerException class sets the **ErrorCode** property to the last client error that occurred.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetDialer (in SocketTools.InternetDialer.dll)

See Also

[InternetDialerException Members](#) | [SocketTools Namespace](#)

InternetDialerException Members

[InternetDialerException overview](#)

Public Instance Constructors

 InternetDialerException	Overloaded. Initializes a new instance of the InternetDialerException class.

Public Instance Properties

ErrorCode	Gets a value which specifies the error that caused the exception.
HelpLink (inherited from Exception)	Gets or sets a link to the help file associated with this exception.
InnerException (inherited from Exception)	Gets the Exception instance that caused the current exception.
Message	Gets a value which describes the error that caused the exception.
Number	Gets a value which specifies the numeric value of the error that caused the exception.
Source (inherited from Exception)	Gets or sets the name of the application or the object that causes the error.
StackTrace (inherited from Exception)	Gets a string representation of the frames on the call stack at the time the current exception was thrown.
TargetSite (inherited from Exception)	Gets the method that throws the current exception.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetBaseException (inherited from Exception)	When overridden in a derived class, returns the Exception that is the root cause of one or more subsequent exceptions.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetObjectData (inherited from Exception)	When overridden in a derived class, sets the SerializationInfo with information about the exception.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString	Creates and returns a string representation of the current exception.

Protected Instance Properties

 HResult (inherited from Exception)	Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetDialerException Class](#) | [SocketTools Namespace](#)

InternetDialerException Constructor

Initializes a new instance of the [InternetDialerException](#) class with the last network error code.

Overload List

Initializes a new instance of the [InternetDialerException](#) class with the last network error code.

```
public InternetDialerException();
```

Initializes a new instance of the [InternetDialerException](#) class with a specified error number.

```
public InternetDialerException(int);
```

Initializes a new instance of the [InternetDialerException](#) class with a specified error message.

```
public InternetDialerException(string);
```

Initializes a new instance of the [InternetDialerException](#) class with a specified error message and a reference to the inner exception that is the cause of this exception.

```
public InternetDialerException(string,Exception);
```

See Also

[InternetDialerException Class](#) | [SocketTools Namespace](#)

InternetDialerException Constructor ()

Initializes a new instance of the [InternetDialerException](#) class with the last network error code.

[Visual Basic]

```
Overloads Public Sub New()
```

[C#]

```
public InternetDialerException();
```

Remarks

The ctor constructor sets the **ErrorCode** property to the last client error that occurred. For more information about the errors that may occur, refer to the [InternetDialer.ErrorCode](#) enumeration.

See Also

[InternetDialerException Class](#) | [SocketTools Namespace](#) | [InternetDialerException Constructor Overload List](#)

InternetDialerException Constructor (String)

Initializes a new instance of the [InternetDialerException](#) class with a specified error message.

[Visual Basic]

```
Overloads Public Sub New( _  
    ByVal message As String _  
)
```

[C#]

```
public InternetDialerException(  
    string message  
);
```

Parameters

message

The error message that explains the reason for the exception.

Remarks

The content of the *message* parameter is intended to be understood by humans. The caller of this constructor is required to ensure that this string has been localized for the current system culture.

See Also

[InternetDialerException Class](#) | [SocketTools Namespace](#) | [InternetDialerException Constructor Overload List](#)

InternetDialerException Constructor (String, Exception)

Initializes a new instance of the [InternetDialerException](#) class with a specified error message and a reference to the inner exception that is the cause of this exception.

[Visual Basic]

```
Overloads Public Sub New( _  
    ByVal message As String, _  
    ByVal innerException As Exception _  
)
```

[C#]

```
public InternetDialerException(  
    string message,  
    Exception innerException  
);
```

Parameters

message

The error message that explains the reason for the exception.

innerException

The exception that is the cause of the current exception. If the *innerException* parameter is not a null reference, the current exception is raised in a **catch** block that handles the inner exception.

Remarks

The content of the message parameter is intended to be understood by humans. The caller of this constructor is required to ensure that this string has been localized for the current system culture.

An exception that is thrown as a direct result of a previous exception should include a reference to the previous exception in the **InnerException** property. The **InnerException** property returns the same value that is passed into the constructor, or a null reference if the **InnerException** property does not supply the inner exception value to the constructor.

See Also

[InternetDialerException Class](#) | [SocketTools Namespace](#) | [InternetDialerException Constructor Overload List](#)

InternetDialerException Constructor (Int32)

Initializes a new instance of the [InternetDialerException](#) class with a specified error number.

```
[Visual Basic]
Overloads Public Sub New( _
    ByVal code As Integer _
)

[C#]
public InternetDialerException(
    int code
);
```

Parameters

code

An integer value which specifies an error code.

Remarks

This constructor sets the **ErrorCode** property to the specified error code. For more information about the errors that may occur, refer to the [InternetDialer.ErrorCode](#) enumeration.

See Also

[InternetDialerException Class](#) | [SocketTools Namespace](#) | [InternetDialerException Constructor Overload List](#)

InternetDialerException Properties

The properties of the **InternetDialerException** class are listed below. For a complete list of **InternetDialerException** class members, see the [InternetDialerException Members](#) topic.

Public Instance Properties

ErrorCode	Gets a value which specifies the error that caused the exception.
HelpLink (inherited from Exception)	Gets or sets a link to the help file associated with this exception.
InnerException (inherited from Exception)	Gets the Exception instance that caused the current exception.
Message	Gets a value which describes the error that caused the exception.
Number	Gets a value which specifies the numeric value of the error that caused the exception.
Source (inherited from Exception)	Gets or sets the name of the application or the object that causes the error.
StackTrace (inherited from Exception)	Gets a string representation of the frames on the call stack at the time the current exception was thrown.
TargetSite (inherited from Exception)	Gets the method that throws the current exception.

Protected Instance Properties

 HRESULT (inherited from Exception)	Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

See Also

[InternetDialerException Class](#) | [SocketTools Namespace](#)

InternetDialerException.ErrorCode Property

Gets a value which specifies the error that caused the exception.

[Visual Basic]

```
Public ReadOnly Property ErrorCode As ErrorCode
```

[C#]

```
public InternetDialer.ErrorCode ErrorCode {get;}
```

Property Value

Returns a [InternetDialer.ErrorCode](#) enumeration value which specifies the error code.

Remarks

The **ErrorCode** property returns the error code that specifies the cause of the exception.

The default constructor for the [InternetDialerException](#) class sets the error code to the last network error that occurred. For more information about the errors that may occur, refer to the [InternetDialer.ErrorCode](#) enumeration.

See Also

[InternetDialerException Class](#) | [SocketTools Namespace](#)

InternetDialerException.Message Property

Gets a value which describes the error that caused the exception.

[Visual Basic]

Overrides Public ReadOnly Property Message As String

[C#]

public override string Message {get;}

Property Value

A string which describes the error that caused the exception.

Remarks

The **Message** property returns a string which describes the error that caused the exception.

See Also

[InternetDialerException Class](#) | [SocketTools Namespace](#)

InternetDialerException.Number Property

Gets a value which specifies the numeric value of the error that caused the exception.

[Visual Basic]

```
Public ReadOnly Property Number As Integer
```

[C#]

```
public int Number {get;}
```

Property Value

An integer value that specifies the error that caused the exception.

Remarks

The **Number** property returns an integer value which specifies the numeric value of the error that caused the exception. This value is the same as the values returned by the Windows Sockets API. For more information about socket error codes, see the Windows Socket Version 2 API error code documentation in MSDN.

See Also

[InternetDialerException Class](#) | [SocketTools Namespace](#)

InternetDialerException Methods

The methods of the **InternetDialerException** class are listed below. For a complete list of **InternetDialerException** class members, see the [InternetDialerException Members](#) topic.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetBaseException (inherited from Exception)	When overridden in a derived class, returns the Exception that is the root cause of one or more subsequent exceptions.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetObjectData (inherited from Exception)	When overridden in a derived class, sets the SerializationInfo with information about the exception.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString	Creates and returns a string representation of the current exception.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetDialerException Class](#) | [SocketTools Namespace](#)

InternetDialerException.ToString Method

Creates and returns a string representation of the current exception.

[Visual Basic]

Overrides Public Function ToString() As String

[C#]

public override string ToString();

Return Value

A string representation of the current exception.

Remarks

The **ToString** method returns a representation of the current exception that is intended to be understood by humans. Where the exception contains culture-sensitive data, the string representation returned by **ToString** is required to take into account the current system culture. Although there are no exact requirements for the format of the returned string, it should attempt to reflect the value of the object as perceived by the user.

This implementation of **ToString** obtains the numeric error code value and a description of the error that caused the current exception. If there is no error message or it is an empty string, then no error message is returned.

This method overrides **ApplicationException.ToString**.

See Also

[InternetDialerException Class](#) | [SocketTools Namespace](#)

InternetServer Class

A general purpose class for developing Internet server applications.

For a list of all members of this type, see [InternetServer Members](#).

System.Object

SocketTools.InternetServer

[Visual Basic]

```
Public Class InternetServer
    Implements IDisposable
```

[C#]

```
public class InternetServer : IDisposable
```

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

The **InternetServer** class provides a simplified interface for creating event-driven, multithreaded server applications using the TCP/IP protocol. The class interface is similar to the **SocketWrench** class, however it is designed specifically to make it easier to implement a server application without requiring the need to manage multiple socket classes. In addition, the **InternetServer** class supports secure communications using the Transport Layer Security (TLS) protocol.

Each instance of the class represents a server, and each active client connection is managed internally and referenced by an integer value which uniquely identifies the client session. All interaction with the server and the clients connected to it uses an event-driven model, with the server application written to respond to events such as **OnConnect**, **OnRead** and **OnWrite**.

Developers who have used the **SocketWrench** class will find the **InternetServer** class has a familiar interface, with a subset of properties and methods that are specific to creating a server application. Each of the network events have an extra parameter which specifies the socket handle which should be used when communicating with the client. This enables the application to communicate with multiple clients without having to create multiple socket classes.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[InternetServer Members](#) | [SocketTools Namespace](#)

InternetServer Members

[InternetServer overview](#)

Public Instance Constructors

 InternetServer Constructor	Initializes a new instance of the InternetServer class.

Public Instance Fields

AdapterAddress	Returns the IP address associated with the specified network adapter.
ClientHandle	Return the socket handle associated with a specific client session.

Public Instance Properties

AdapterCount	Get the number of available local and remote network adapters.
Backlog	Gets and sets the number of client connections that may be queued by the server.
CertificateName	Gets and sets a value that specifies the name of the server certificate.
CertificatePassword	Gets and sets the password associated with the server certificate.
CertificateStore	Gets and sets a value that specifies the name of the local certificate store.
CertificateUser	Gets and sets the user that owns the server certificate.
ClientAddress	Gets a value that specifies the Internet address of the current client session.
ClientCount	Gets the number of active client sessions connected to the server.
ClientHost	Gets a value that specifies the hostname for the current client session.
ClientId	Gets the unique client identifier for the current client session.
ClientName	Gets and sets a unique string moniker that is associated with the current client session.
ClientPort	Gets a value that specifies the port number used by the current client session.
ClientThread	Gets the thread ID for the current client session.
CodePage	Gets and sets the code page used when reading and writing text.
ExternalAddress	Gets a value that specifies the external Internet address for the local system.

IdleTime	Gets a value which specifies the amount of time a socket has been idle
IsActive	Gets a value which indicates if the server is active.
IsBlocked	Gets a value which indicates if the current thread is performing a blocking socket operation.
IsClosed	Gets a value which indicates if the connection to the client has been closed.
IsInitialized	Gets a value which indicates if the current instance of the class has been initialized successfully.
IsListening	Gets a value which indicates if the server is listening for client connections.
IsLocked	Gets a value which indicates if the server has been locked.
IsReadable	Gets a value which indicates if there is data available to be read from the current client.
IsWritable	Gets a value which indicates if data can be written to the current client without blocking.
KeepAlive	Gets and sets a value which indicates if keep-alive packets are sent on a connected socket.
LastError	Gets and sets a value which specifies the last error that has occurred.
LastErrorString	Gets a value which describes the last error that has occurred.
MaxClients	Gets and sets the maximum number of clients that can connect to the server.
MemoryUsage	Gets the amount of unmanaged memory currently allocated by the server.
NoDelay	Gets and sets a value which specifies if the Nagle algorithm should be enabled or disabled.
Options	Gets and sets a value which specifies one or more server options.
Priority	Gets and sets a value which specifies the server priority.
ReuseAddress	Gets and sets a value which indicates if the server address can be reused.
Secure	Gets and sets a value which specifies if client connections are secure.
SecureProtocol	Gets and sets a value which specifies the protocol used for secure client connections.
ServerAddress	Gets and sets the address that will be used by the server to listen for connections.
ServerHandle	Gets the handle to the socket created to listen for

	client connections.
ServerName	Gets a value which specifies the host name for the local system.
ServerPort	Gets and sets the port number that will be used by the server to listen for connections.
ServerThread	Gets the thread ID for the current server.
StackSize	Gets and sets the size of the stack allocated for threads created by the server.
Status	Gets a value which specifies the current status of the server.
ThrowError	Gets and sets a value which specifies if method calls should throw exceptions when an error occurs.
Timeout	Gets and sets a value which specifies a timeout period in seconds.
Trace	Gets and sets a value which indicates if network function logging is enabled.
TraceFile	Gets and sets a value which specifies the name of the network function tracing logfile.
TraceFlags	Gets and sets a value which specifies the network function tracing flags.
Version	Gets a value which returns the current version of the InternetServer class library.

Public Instance Methods

Abort	Overloaded. Abort the connection with a remote host.
Broadcast	Overloaded. Broadcast data to all active clients connected to the server
Cancel	Overloaded. Cancel the current blocking socket operation.
Disconnect	Overloaded. Disconnect the specified client connection from the server.
Dispose	Overloaded. Releases all resources used by InternetServer .
Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
FindClient	Overloaded. Return the socket handle for the client session with the specified moniker.
Flush	Overloaded. Flush the contents of the send and receive buffers for the client socket.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.

GetType (inherited from Object)	Gets the Type of the current instance.
Initialize	Overloaded. Initialize an instance of the InternetServer class.
Lock	Lock the server to synchronize access to shared data for all active client sessions.
Peek	Overloaded. Read data from the client and store it in a byte array, but do not remove the data from the socket buffers.
Read	Overloaded. Read data from the client socket and store it in a byte array.
ReadLine	Overloaded. Read up to a line of data from the client and return it in a string buffer.
Reject	Overloaded. Rejects a connection request from a client.
Reset	Reset the internal state of the object, resetting all properties to their default values.
Resolve	Resolves a host name to a host IP address.
Restart	Restarts the server and terminates all active client connections.
Resume	Resume accepting new client connections.
Start	Overloaded. Start listening for client connections on the specified IP address and port number.
Stop	Stop listening for new client connections and terminate all active clients already connected to the server.
Suspend	Overloaded. Suspend accepting new client connections with additional options.
Throttle	Overloaded. Limit the maximum number of client connections, connections per IP address and connection rate.
ToString (inherited from Object)	Returns a String that represents the current Object.
Uninitialize	Uninitialize the class library and release any resources allocated for the server.
Unlock	Unlock the server and allow other server threads to resume execution.
Write	Overloaded. Write one or more bytes of data to a client.
WriteLine	Overloaded. Send a line of text to a client, terminated by a carriage-return and linefeed.

Public Instance Events

OnAccept	Occurs when a client attempts to establish a

	connection with the server.
OnCancel	Occurs when a blocking socket operation is canceled.
OnConnect	Occurs when a connection is established with the remote host.
OnDisconnect	Occurs when the remote host disconnects from the local system.
OnError	Occurs when an socket operation fails.
OnIdle	Occurs when there are no clients connected to the server.
OnRead	Occurs when data is available to be read from the client.
OnStart	Occurs when the server starts accepting connections.
OnStop	Occurs when the server stops accepting connections.
OnTimeout	Occurs when a blocking operation fails to complete before the timeout period elapses.
OnWrite	Occurs when data can be written to the client.

Protected Instance Methods

 Dispose	Overloaded. Releases the unmanaged resources allocated by the InternetServer class and optionally releases the managed resources.
 Finalize	Destroys an instance of the class, releasing the resources allocated for the session and unloading the networking library.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer Constructor

Initializes a new instance of the InternetServer class.

```
[Visual Basic]  
Public Sub New()
```

```
[C#]  
public InternetServer();
```

Example

The following example demonstrates creating an instance of the **InternetServer** class object and starting a server using the [Start](#) method.

```
Dim Server As SocketTools.InternetServer  
Dim strLocalAddress As String  
Dim nLocalPort As Integer  
  
Server = New SocketTools.InternetServer  
  
strLocalAddress = TextBox1.Text.Trim()  
nLocalPort = Val(TextBox2.Text)  
  
If Server.Start(strLocalAddress, nLocalPort) Then  
    StatusBar1.Text = "The server has started listening for connections"  
Else  
    StatusBar1.Text = "The server could not be started"  
End If
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer Fields

The fields of the **InternetServer** class are listed below. For a complete list of **InternetServer** class members, see the [InternetServer Members](#) topic.

Public Instance Fields

AdapterAddress	Returns the IP address associated with the specified network adapter.
ClientHandle	Return the socket handle associated with a specific client session.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.AdapterAddress Field

Returns the IP address associated with the specified network adapter.

[Visual Basic]

Public **ReadOnly** **AdapterAddress** As [AdapterAddressArray](#)

[C#]

public **readonly** [AdapterAddressArray](#) **AdapterAddress**;

Remarks

The **AdapterAddress** array returns the IP addresses that are associated with the local network or remote dial-up network adapters configured on the system. The **AdapterCount** property can be used to determine the number of adapters that are available.

Multihomed systems with more than one local network adapter, or a combination of local and dial-up adapters will not be listed in a specific order. An application should not make the assumption that the first address returned by **AdapterAddress** always refers to a local network adapter.

Note that it is possible that the **AdapterCount** property will return 0, and **AdapterAddress** will return an empty string. This indicates that the system does not have a physical network adapter with an assigned IP address, and there are no dial-up networking connections currently active. If a dial-up networking connection is established at some later point, the **AdapterCount** property will change to 1, and the **AdapterAddress** property will return the IP address allocated for that connection.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [AdapterAddressArray Class](#) | [AdapterCount Property](#)

InternetServer.AdapterAddressArray Class

The class used to return the network adapters configured for the local host.

For a list of all members of this type, see [InternetServer.AdapterAddressArray Members](#).

System.Object

SocketTools.InternetServer.AdapterAddressArray

[Visual Basic]

```
<DefaultMember(MemberName:="Item")>  
Public Class InternetServer.AdapterAddressArray
```

[C#]

```
[DefaultMember(MemberName="Item")]  
public class InternetServer.AdapterAddressArray
```

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

The **AdapterAddressArray** class returns the network adapters configured for the local host. A read-only instance of this class is created by the InternetServer class constructor and accessed using the **AdapterAddress** array.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[InternetServer.AdapterAddressArray Members](#) | [SocketTools Namespace](#) | [AdapterAddress Field](#)

InternetServer.AdapterAddressArray Members

[InternetServer.AdapterAddressArray overview](#)

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Returns the address for the specified network adapter.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetServer.AdapterAddressArray Class](#) | [SocketTools Namespace](#) | [AdapterAddress Field](#)

InternetServer.AdapterAddressArray Properties

The properties of the **InternetServer.AdapterAddressArray** class are listed below. For a complete list of **InternetServer.AdapterAddressArray** class members, see the [InternetServer.AdapterAddressArray Members](#) topic.

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Returns the address for the specified network adapter.

See Also

[InternetServer.AdapterAddressArray Class](#) | [SocketTools Namespace](#) | [AdapterAddress Field](#)

InternetServer.AdapterAddressArray.Count Property

Returns the maximum number of elements in the array.

[Visual Basic]

Public ReadOnly Property Count As Integer

[C#]

public int Count {get;}

Property Value

An integer value.

Remarks

This property will return the number of network adapters that are configured for the local system.

See Also

[InternetServer.AdapterAddressArray Class](#) | [SocketTools Namespace](#)

InternetServer.AdapterAddressArray.Item Property

Returns the address for the specified network adapter.

[Visual Basic]

```
Public Default ReadOnly Property Item( _  
    ByVal index As Integer _  
) As String
```

[C#]

```
public string this[  
    int index  
] {get;}
```

Parameters

index

An integer value which specifies the index into the array.

Property Value

A string which specifies an Internet address using dot notation.

See Also

[InternetServer.AdapterAddressArray Class](#) | [SocketTools Namespace](#)

InternetServer.ClientHandle Field

Return the socket handle associated with a specific client session.

[Visual Basic]

```
Public ReadOnly ClientHandle As ClientHandleArray
```

[C#]

```
public readonly ClientHandleArray ClientHandle;
```

Remarks

The **ClientHandle** array is a read-only, zero-based property array that returns the socket handle allocated for the client session specified by the Index parameter. An exception will be thrown if the index value exceeds the maximum number of active client sessions. To determine the number of clients that are currently connected to the server, use the **ClientCount** property.

You should always check the value of the **ClientCount** property prior to enumerating through the client connections using the **ClientHandle** array. Never assume that a particular client session will always be found in the same position in the array. The socket handles returned by the array can be used in conjunction with the **Read** and **Write** methods to exchange data with a particular client session outside of an event handler.

The index into this array may also be a string which specifies the name of a client session, as set by the **ClientName** property. This can be a convenient way to obtain the socket handle for a specific client by name, rather than a numeric index.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.ClientHandleArray Class

The class used to return the handle to client sessions established with the server.

For a list of all members of this type, see [InternetServer.ClientHandleArray Members](#).

System.Object

SocketTools.InternetServer.ClientHandleArray

[Visual Basic]

```
<DefaultMember(MemberName:="Item")>  
Public Class InternetServer.ClientHandleArray
```

[C#]

```
[DefaultMember(MemberName="Item")]  
public class InternetServer.ClientHandleArray
```

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

The **ClientHandleArray** class returns the handles for each client session that has been established with the server. A read-only instance of this class is created by the InternetServer class constructor and accessed using the **ClientHandle** array.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[InternetServer.ClientHandleArray Members](#) | [SocketTools Namespace](#) | [ClientHandle Field](#)

InternetServer.ClientHandleArray Members

[InternetServer.ClientHandleArray overview](#)

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Overloaded. Returns the socket handle for the specified client session.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetServer.ClientHandleArray Class](#) | [SocketTools Namespace](#) | [ClientHandle Field](#)

InternetServer.ClientHandleArray Properties

The properties of the **InternetServer.ClientHandleArray** class are listed below. For a complete list of **InternetServer.ClientHandleArray** class members, see the [InternetServer.ClientHandleArray Members](#) topic.

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Overloaded. Returns the socket handle for the specified client session.

See Also

[InternetServer.ClientHandleArray Class](#) | [SocketTools Namespace](#) | [ClientHandle Field](#)

InternetServer.ClientHandleArray.Count Property

Returns the maximum number of elements in the array.

[Visual Basic]

Public ReadOnly Property Count As Integer

[C#]

public int Count {get;}

Property Value

An integer value.

Remarks

This property will return the number of client sessions which have been established with the server.

See Also

[InternetServer.ClientHandleArray Class](#) | [SocketTools Namespace](#)

InternetServer.ClientHandleArray.Item Property

Returns the socket handle for the specified client session.

Overload List

Returns the socket handle for the specified client session.

```
public int this[int] {get;}
```

Returns the socket handle for the specified client session.

```
public int this[string] {get;}
```

See Also

[InternetServer.ClientHandleArray Class](#) | [SocketTools Namespace](#)

InternetServer.ClientHandleArray.Item Property (Int32)

Returns the socket handle for the specified client session.

[Visual Basic]

```
Overloads Public Default ReadOnly Property Item( _  
    ByVal index As Integer _  
) As Integer
```

[C#]

```
public int this[  
    int index  
] {get;}
```

Property Value

An integer value which specifies the socket handle for the client session.

See Also

[InternetServer.ClientHandleArray Class](#) | [SocketTools Namespace](#) | [InternetServer.ClientHandleArray.Item Overload List](#)

InternetServer.ClientHandleArray.Item Property (String)

Returns the socket handle for the specified client session.

[Visual Basic]

```
Overloads Public Default ReadOnly Property Item( _  
    ByVal index As String _  
) As Integer
```

[C#]

```
public int this[  
    string index  
] {get;}
```

Property Value

A string value which specifies the name of a client session.

See Also

[InternetServer.ClientHandleArray Class](#) | [SocketTools Namespace](#) | [InternetServer.ClientHandleArray.Item Overload List](#)

InternetServer Properties

The properties of the **InternetServer** class are listed below. For a complete list of **InternetServer** class members, see the [InternetServer Members](#) topic.

Public Instance Properties

AdapterCount	Get the number of available local and remote network adapters.
Backlog	Gets and sets the number of client connections that may be queued by the server.
CertificateName	Gets and sets a value that specifies the name of the server certificate.
CertificatePassword	Gets and sets the password associated with the server certificate.
CertificateStore	Gets and sets a value that specifies the name of the local certificate store.
CertificateUser	Gets and sets the user that owns the server certificate.
ClientAddress	Gets a value that specifies the Internet address of the current client session.
ClientCount	Gets the number of active client sessions connected to the server.
ClientHost	Gets a value that specifies the hostname for the current client session.
ClientId	Gets the unique client identifier for the current client session.
ClientName	Gets and sets a unique string moniker that is associated with the current client session.
ClientPort	Gets a value that specifies the port number used by the current client session.
ClientThread	Gets the thread ID for the current client session.
CodePage	Gets and sets the code page used when reading and writing text.
ExternalAddress	Gets a value that specifies the external Internet address for the local system.
IdleTime	Gets a value which specifies the amount of time a socket has been idle
IsActive	Gets a value which indicates if the server is active.
IsBlocked	Gets a value which indicates if the current thread is performing a blocking socket operation.
IsClosed	Gets a value which indicates if the connection to the client has been closed.

IsInitialized	Gets a value which indicates if the current instance of the class has been initialized successfully.
IsListening	Gets a value which indicates if the server is listening for client connections.
IsLocked	Gets a value which indicates if the server has been locked.
IsReadable	Gets a value which indicates if there is data available to be read from the current client.
IsWritable	Gets a value which indicates if data can be written to the current client without blocking.
KeepAlive	Gets and sets a value which indicates if keep-alive packets are sent on a connected socket.
LastError	Gets and sets a value which specifies the last error that has occurred.
LastErrorString	Gets a value which describes the last error that has occurred.
MaxClients	Gets and sets the maximum number of clients that can connect to the server.
MemoryUsage	Gets the amount of unmanaged memory currently allocated by the server.
NoDelay	Gets and sets a value which specifies if the Nagle algorithm should be enabled or disabled.
Options	Gets and sets a value which specifies one or more server options.
Priority	Gets and sets a value which specifies the server priority.
ReuseAddress	Gets and sets a value which indicates if the server address can be reused.
Secure	Gets and sets a value which specifies if client connections are secure.
SecureProtocol	Gets and sets a value which specifies the protocol used for secure client connections.
ServerAddress	Gets and sets the address that will be used by the server to listen for connections.
ServerHandle	Gets the handle to the socket created to listen for client connections.
ServerName	Gets a value which specifies the host name for the local system.
ServerPort	Gets and sets the port number that will be used by the server to listen for connections.
ServerThread	Gets the thread ID for the current server.
StackSize	Gets and sets the size of the stack allocated for

	threads created by the server.
Status	Gets a value which specifies the current status of the server.
ThrowError	Gets and sets a value which specifies if method calls should throw exceptions when an error occurs.
Timeout	Gets and sets a value which specifies a timeout period in seconds.
Trace	Gets and sets a value which indicates if network function logging is enabled.
TraceFile	Gets and sets a value which specifies the name of the network function tracing logfile.
TraceFlags	Gets and sets a value which specifies the network function tracing flags.
Version	Gets a value which returns the current version of the InternetServer class library.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.AdapterCount Property

Get the number of available local and remote network adapters.

[Visual Basic]

Public ReadOnly Property AdapterCount As Integer

[C#]

public int AdapterCount {get;}

Property Value

Returns the number of available local and remote network adapters.

Remarks

The **AdapterCount** property returns the number of local and remote dial-up networking adapters available on the local system. This value can be used in conjunction with the **AdapterAddress** array to enumerate the IP addresses assigned to the various network adapters.

Note that it is possible that the **AdapterCount** property will return 0, and **AdapterAddress** will return an empty string. This indicates that the system does not have a physical network adapter with an assigned IP address, and there are no dial-up networking connections currently active. If a dial-up networking connection is established at some later point, the **AdapterCount** property will change to 1, and the **AdapterAddress** property will return the IP address allocated for that connection.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [AdapterAddress Field](#)

InternetServer.Backlog Property

Gets and sets the number of client connections that may be queued by the server.

[Visual Basic]

Public Property Backlog As Integer

[C#]

public int Backlog {get; set;}

Property Value

Returns an integer value that specifies the size of the backlog queue. The default value is 5.

Remarks

The **Backlog** property specifies the maximum size of the queue used to manage pending connections to the service. If the property is set to value which exceeds the maximum size for the underlying service provider, it will be silently adjusted to the nearest legal value. There is no standard way to determine what the maximum backlog value is.

This property should be set to the desired value before the **Start** method is called. The default backlog value is 5 on all Windows platforms. The Windows Server platforms support a maximum backlog value of 200.

Note that this property does not specify the total number of connections that the server application may accept. It only specifies the size of the backlog queue which is used to manage pending client connections. Once the client connection has been accepted, it is removed from the queue. Set the **MaxClients** property to specify the maximum number of clients that may connect with the server.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [IsListening Property](#) | [MaxClients Property](#) | [Start Method](#)

InternetServer.CertificateName Property

Gets and sets a value that specifies the name of the server certificate.

[Visual Basic]

```
Public Property CertificateName As String
```

[C#]

```
public string CertificateName {get; set;}
```

Property Value

A string which specifies the server certificate name.

Remarks

The **CertificateName** property sets the common name or friendly name of the certificate that should be used when starting a secure server. If the **Secure** property is set to True, this property must specify a valid certificate name. The certificate must have a private key associated with it, otherwise client connections will fail because the class will be unable to create a security context for the session.

When the certificate store is searched for a matching certificate, it will first search for any certificate with a friendly name that matches the property value. If no valid certificate is found, it will then search for a certificate with a matching common name.

Certificates may be installed and viewed on the local system using the Certificate Manager that is included with the Windows operating system. For more information, refer to the documentation for the Microsoft Management Console.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [CertificateStore Property](#) | [Secure Property](#)

InternetServer.CertificateStore Property

Gets and sets a value that specifies the name of the local certificate store.

[Visual Basic]

```
Public Property CertificateStore As String
```

[C#]

```
public string CertificateStore {get; set;}
```

Property Value

A string which specifies the certificate store name. The default value is the current user's personal certificate store.

Remarks

The **CertificateStore** property is used to specify the name of the certificate store which contains the security certificate to use when security is enabled for the server. The certificate may either be stored in the registry or in a file. If the certificate is stored in the registry, then this property should be set to one of the following predefined values:

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. If a certificate store is not specified, this is the default value that is used.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

In most cases the server certificate will be installed in the user's personal certificate store, and therefore it is not necessary to set this property value because that is the default location that will be used to search for the certificate. This property is only used if the **CertificateName** property is also set to a valid certificate name.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU" for the current user, or "HKLM" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, it will default to the certificate store for the current user.

This property may also be used to specify a file that contains the server certificate. In this case, the property should specify the full path to the file and must contain both the certificate and private key in PKCS #12 format. If the file is protected by a password, the **CertificatePassword** property must also be set to specify the password.

This property may also be used to specify a file that contains the server certificate. In this case, the property should specify the full path to the file and must contain both the certificate and private key in PKCS #12 format. If the file is protected by a password, the **CertificatePassword** property must also be set to specify the password.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [CertificatePassword Property](#) | [Secure Property](#)

InternetServer.ClientAddress Property

Gets a value that specifies the Internet address of the current client session.

[Visual Basic]

```
Public ReadOnly Property ClientAddress As String
```

[C#]

```
public string ClientAddress {get;}
```

Property Value

A string which specifies an Internet address in dotted notation.

Remarks

The **ClientAddress** property returns the address of the current client session which has connected to the server. This property only returns a meaningful value inside an event handler such as **OnAccept** or **OnConnect**.

If this property is accessed inside an **OnAccept** event handler, it will return the address of the client that is requesting the connection. The server application may use this information to determine if it wishes to accept or reject the client connection.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [ClientHost Property](#) | [ClientPort Property](#)

InternetServer.ClientCount Property

Gets the number of active client sessions connected to the server.

[Visual Basic]

```
Public ReadOnly Property ClientCount As Integer
```

[C#]

```
public int ClientCount {get;}
```

Property Value

An integer value which specifies the number of active client sessions.

Remarks

The **ClientCount** read-only property returns the number of active client sessions that have been established with the server. This property is typically used in conjunction with the **ClientHandle** array to enumerate the handles for all client sessions.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.ClientHost Property

Gets a value that specifies the hostname for the current client session.

[Visual Basic]

```
Public ReadOnly Property ClientHost As String
```

[C#]

```
public string ClientHost {get;}
```

Property Value

A string which specifies the peer host name.

Remarks

The **ClientHost** property returns the hostname of the current client session which has established a connection with the server. This property value is only meaningful when accessed within an event handler, such as the **OnConnect** event.

Accessing this property causes the class to perform a blocking reverse DNS lookup, attempting to match the client Internet address with a hostname. Not all addresses have a reverse DNS record, in which case this property will return an empty string. It is recommended that most applications use the value of the **ClientAddress** property rather than use the **ClientHost** property to distinguish between client connections.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [ClientAddress Property](#) | [ClientPort Property](#)

InternetServer.ClientId Property

Gets the unique client identifier for the current client session.

[Visual Basic]

```
Public ReadOnly Property ClientId As Integer
```

[C#]

```
public int ClientId {get;}
```

Property Value

An integer value which uniquely identifies the client session.

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value. This value can be used by the application to identify that client session, and is different than the socket handle allocated for the client. While it is possible for a client socket handle to be reused by the operating system, client IDs are unique throughout the life of the server session and are never duplicated.

It is important to note that the actual value of the client ID should be considered opaque. It is only guaranteed that the value will be greater than zero, and that it will be unique to the client session.

This property only returns a meaningful value when accessed from within a class event handler, or a method that has been invoked from within an event handler.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.ClientName Property

Gets and sets a unique string moniker that is associated with the current client session.

[Visual Basic]

```
Public Property ClientName As String
```

[C#]

```
public string ClientName {get; set;}
```

Property Value

A string moniker which uniquely identifies the client session. If no moniker has been specified for the client session, this property will return an empty string.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session by setting the **ClientName** property from within a class event handler such as the **OnConnect** event.

Monikers are not case-sensitive, and they must be unique so that no client socket for a particular server can have the same moniker. The maximum length for a moniker is 127 characters.

This property only returns a meaningful value when accessed from within a class event handler, or a method that has been invoked from within an event handler.

Example

The following example sets the moniker for the client session in the [OnConnect](#) event handler.

```
private void Server1_OnConnect(object sender,
SocketTools.InternetServer.ConnectEventArgs e)
{
    Server1.ClientName = "Client" + Server1.ClientId.ToString();
}
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.ClientPort Property

Gets a value that specifies the port number used by the current client session.

[Visual Basic]

```
Public ReadOnly Property ClientPort As Integer
```

[C#]

```
public int ClientPort {get;}
```

Property Value

An integer value which specifies the peer port number.

Remarks

The **ClientPort** property returns the port number that the current client has used when establishing a connection with the server. This property value is only meaningful when accessed within an event handler such as the **OnConnect** event.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.ClientThread Property

Gets the thread ID for the current client session.

[Visual Basic]

```
Public ReadOnly Property ClientThread As Integer
```

[C#]

```
public int ClientThread {get;}
```

Property Value

An integer value which identifies the client thread that was created to manage the client session.

Remarks

Until the thread terminates, the thread identifier uniquely identifies the thread throughout the system.

This property only returns a meaningful value when accessed from within a class event handler, or a method that has been invoked from within an event handler.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.ExternalAddress Property

Gets a value that specifies the external Internet address for the local system.

[Visual Basic]

```
Public ReadOnly Property ExternalAddress As String
```

[C#]

```
public string ExternalAddress {get;}
```

Property Value

A string which specifies an Internet address using dotted notation.

Remarks

The **ExternalAddress** property returns the IP address assigned to the router that connects the local host to the Internet. This is typically used by an application executing on a system in a local network that uses a router which performs Network Address Translation (NAT). The **ExternalAddress** property can be used to determine the IP address assigned to the router on the Internet side of the connection and can be particularly useful for servers running on a system behind a NAT router.

Using this property requires that you have an active connection to the Internet; checking the value of this property on a system that uses dial-up networking may cause the operating system to automatically connect to the Internet service provider. The class may be unable to determine the external IP address for the local host for a number of reasons, particularly if the system is behind a firewall or uses a proxy server that restricts access to external sites on the Internet. If the external address for the local host cannot be determined, the property will return an empty string.

If the class is able to obtain a valid external address for the local host, that address will be cached for sixty minutes. Because dial-up connections typically have different IP addresses assigned to them each time the system is connected to the Internet, it is recommended that this property only be used in conjunction with broadband connections using a NAT router.

It is important to note that checking this property value may cause the current thread to block until the external IP address can be resolved.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.IsBlocked Property

Gets a value which indicates if the current thread is performing a blocking socket operation.

[Visual Basic]

Public ReadOnly Property IsBlocked As Boolean

[C#]

public bool IsBlocked {get;}

Property Value

Returns **true** if the current thread is blocking, otherwise returns **false**.

Remarks

The **IsBlocked** property returns **true** if the current thread is blocked performing an operation. Because the Windows Sockets API only permits one blocking operation per thread of execution, this property should be checked before starting any blocking operation in response to an event.

If the **IsBlocked** property returns **false**, this means there are no blocking operations on the current thread at that time. However, this does not guarantee that the next socket operation will not fail. An application should always check the return value from a socket operation and check the value of the **LastError** property if an error occurs.

Note that this property will return **true** if there is any blocking operation being performed by the current thread, regardless of whether this specific instance of the class is responsible for the blocking operation or not.

This property only returns a meaningful value when accessed from within a class event handler, or a method that has been invoked from within an event handler.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.IsClosed Property

Gets a value which indicates if the connection to the client has been closed.

[Visual Basic]

Public ReadOnly Property IsClosed As Boolean

[C#]

public bool IsClosed {get;}

Property Value

Returns **true** if the connection has been closed; otherwise returns **false**.

Remarks

This property only returns a meaningful value when accessed from within a class event handler, or a method that has been invoked from within an event handler.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.IsInitialized Property

Gets a value which indicates if the current instance of the class has been initialized successfully.

[Visual Basic]

Public ReadOnly Property IsInitialized As Boolean

[C#]

public bool IsInitialized {get;}

Property Value

Returns **true** if the class instance has been initialized; otherwise returns **false**.

Remarks

The **IsInitialized** property is used to determine if the current instance of the class has been initialized properly. Normally this is done automatically by the class constructor, however there are circumstances where the class may not be able to initialize itself.

The most common reasons that a class instance may not initialize correctly is that no runtime license key has been defined in the assembly or the license key provided is invalid. It may also indicate a problem with the system configuration or user access rights, such as not being able to load the required networking libraries or not being able to access the system registry.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.IsListening Property

Gets a value which indicates if the server is listening for client connections.

[Visual Basic]

Public ReadOnly Property IsListening As Boolean

[C#]

public bool IsListening {get;}

Property Value

Returns **true** if the server is listening for client connections; otherwise returns **false**.

Remarks

The **IsListening** property will return **true** if the **Start** method was called and the server is currently accepting incoming client connections. In all other situations, this property will return **false**.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.IsLocked Property

Gets a value which indicates if the server has been locked.

[Visual Basic]

```
Public ReadOnly Property IsLocked As Boolean
```

[C#]

```
public bool IsLocked {get;}
```

Property Value

Returns **true** if the server has been locked; otherwise returns **false**.

Remarks

The **IsLocked** property returns **true** if the server has been locked using the **Lock** method. When a server is locked, all background threads created by the server will block, waiting for the lock to be released. If this property returns a value of **true**, no client connections can be accepted by the server, and no network events will be generated.

The **Lock** method creates a critical section which prevents other threads from performing any network operation. This is useful when the program needs to update global data and wants to ensure that no network operations occur while the data is being modified. However, applications must take care to release the lock as quickly as possible. If a function locks the server, it must make sure that it releases the lock before exiting that function. Leaving the server locked across function calls or event handlers can result in the server becoming non-responsive.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [Lock Method](#) | [Unlock Method](#)

InternetServer.IsReadable Property

Gets a value which indicates if there is data available to be read from the current client.

[Visual Basic]

```
Public ReadOnly Property IsReadable As Boolean
```

[C#]

```
public bool IsReadable {get;}
```

Property Value

Returns **true** if there is data available to be read; otherwise returns **false**.

Remarks

The **IsReadable** property returns **true** if data can be read from the current client session without causing the current thread to block. Note that even if this property does return **true** indicating that there is data available to be read, applications should always check the return value from the **Read** method.

This property only returns a meaningful value when accessed from within a class event handler, or a method that has been invoked from within an event handler.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.IsWritable Property

Gets a value which indicates if data can be written to the current client without blocking.

[Visual Basic]

```
Public ReadOnly Property IsWritable As Boolean
```

[C#]

```
public bool IsWritable {get;}
```

Property Value

Returns **true** if data can be written to the client; otherwise returns **false**.

Remarks

The **IsWritable** property returns **true** if data can be written to the client without causing the current thread to block. Note that even if this property does return **true** indicating that data can be written to the client, applications should always check the return value from the **Write** method.

This property only returns a meaningful value when accessed from within a class event handler, or a method that has been invoked from within an event handler.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.KeepAlive Property

Gets and sets a value which indicates if keep-alive packets are sent on a connected socket.

[Visual Basic]

Public Property KeepAlive As Boolean

[C#]

public bool KeepAlive {get; set;}

Property Value

Returns **true** if keep-alive packets are enabled, otherwise returns **false**. The default value is **false**.

Remarks

Setting the **KeepAlive** property to a value of **true** specifies that special packets are to be sent to the remote system when no data is being exchanged to ensure the connection remains active. This property can only be set for sockets that were created with the **Protocol** property set to a value of **SocketProtocol.protocolStream**.

It is important to note that the system will not start generating keep-alive packets until two hours after it has been enabled, so this option is only relevant for connections that will be maintained for long periods of time. The actual interval for the keep-alive period can only be changed in the system registry and affects all sockets, system-wide. For more information, refer to Microsoft Knowledge Base article 314053.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.LastError Property

Gets and sets a value which specifies the last error that has occurred.

[Visual Basic]

```
Public Property LastError As ErrorCode
```

[C#]

```
public InternetServer.ErrorCode LastError {get; set;}
```

Property Value

Returns an [ErrorCode](#) enumeration value which specifies the last error code.

Remarks

The **LastError** property returns the error code associated with the last error that occurred for the current instance of the class. It is important to note that this value only has meaning if the previous method indicates that an error has actually occurred.

It is possible to explicitly clear the last error code by assigning the property to the value **ErrorCode.None**.

The error code value can be cast to an integer value for display purposes if required. For a description of the error that can be displayed using a message box or some other similar mechanism, get the value of the **LastErrorString** property.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.LastErrorString Property

Gets a value which describes the last error that has occurred.

[Visual Basic]

```
Public ReadOnly Property LastErrorString As String
```

[C#]

```
public string LastErrorString {get;}
```

Property Value

A string which describes the last error that has occurred.

Remarks

The **LastErrorString** property can be used to obtain a description of the last error that occurred for the current instance of the class. It is important to note that this value only has meaning if the previous method indicates that an error has actually occurred.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.MaxClients Property

Gets and sets the maximum number of clients that can connect to the server.

[Visual Basic]

Public Property MaxClients As Integer

[C#]

public int MaxClients {get; set;}

Property Value

An integer value which specifies the maximum number of client sessions that will be accepted by the server. A value of zero specifies that there is no fixed limit to the maximum number of clients.

Remarks

The **MaxClients** property specifies the maximum number of client connections that will be accepted by the server. Once the maximum number of connections has been established, the server will reject any subsequent connections until the number of active client connections drops below the specified value. A value of zero specifies that there should be no limit on the number of clients.

Changing the value of this property while a server is actively listening for connections will modify the maximum number of client connections permitted, but it will not affect connections that have already been established.

By default, there are no limits on the number of client connections or the connection rate when a server is started. Use the **Throttle** method to change the maximum number of client connections per IP address or the overall connection rate threshold for the server.

It is important to note that regardless of the maximum number of clients specified by this property, the actual number of client connections that can be managed by the server depends on the number of sockets that can be allocated from the operating system. The amount of physical memory installed on the system affects the number of connections that can be maintained because each connection allocates memory for the socket context from the non-paged memory pool.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.MemoryUsage Property

Gets the amount of unmanaged memory currently allocated by the server.

[Visual Basic]

Public ReadOnly Property MemoryUsage As Long

[C#]

public long MemoryUsage {get;}

Property Value

A long integer which specifies the number of bytes of memory allocated.

Remarks

This read-only property returns the amount of memory allocated by the server and all active client sessions. It enumerates all unmanaged memory allocations made by the server process and client session threads, returning the total number of bytes allocated for the server. This value reflects the amount of memory explicitly allocated by the class and does not reflect the total working set size of the process, or the memory allocated on the managed heap which is used by the .NET garbage collector.

Getting the value of this property forces the server into a locked state, and all client sessions will block while the memory usage is being calculated. Because this enumerates all unmanaged heaps allocated for the server process, it can be an expensive operation, particularly when there are a large number of active clients connected to the server. Frequently checking the value of this property can significantly degrade the performance of the server. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active client sessions. If the value returned by this property remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [StackSize Property](#)

InternetServer.NoDelay Property

Gets and sets a value which specifies if the Nagle algorithm should be enabled or disabled.

[Visual Basic]

Public Property NoDelay As Boolean

[C#]

public bool NoDelay {get; set;}

Property Value

Returns **true** if the Nagle algorithm has been disabled; otherwise it returns **false**. The default value is **false**.

Remarks

The **NoDelay** property is used to enable or disable the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host. By default this property value is set to **false**, which enables the Nagle algorithm (in other words, the data being written may not actually be sent until it is optimal to do so). Setting this property to **true** disables the Nagle algorithm, maintaining the time delays between the data packets being sent.

This property should be set to **true** only if it is absolutely required and the implications of doing so are understood. Disabling the Nagle algorithm can have a significant negative impact on the performance of your server.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Options Property

Gets and sets a value which specifies one or more server options.

[Visual Basic]

Public Property Options As [ServerOptions](#)

[C#]

public [InternetServer.ServerOptions](#) Options {get; set;}

Property Value

Returns one or more [ServerOptions](#) enumeration flags which specify the options for the server. The default value for this property is **OptionNone**.

Remarks

The **Options** property specifies one or more default options which are used when starting the server using the **Start** method.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [Start Method](#)

InternetServer.Priority Property

Gets and sets a value which specifies the server priority.

[Visual Basic]

Public Property Priority As [ServerPriority](#)

[C#]

public [InternetServer.ServerPriority](#) **Priority {get; set;}**

Property Value

Returns a [ServerPriority](#) enumeration value which specifies the current server priority. The default value for this property is **Normal**.

Remarks

The **Priority** property can be used to control the processor usage, memory and network bandwidth allocated by the server for client sessions. The default priority balances resource utilization and client throughput while ensuring that the user interface remains responsive to the user. Lower priorities reduce the overall resource utilization at the expense of throughput.

Higher priority values increases the thread priority and processor utilization for the client sessions. It is not recommended that you increase the server priority unless you understand the implications of doing so and have thoroughly tested your application. Raising the priority of the server can have a negative impact on the responsiveness of the user interface.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [ServerPriority Enumeration](#)

InternetServer.ReuseAddress Property

Gets and sets a value which indicates if the server address can be reused.

[Visual Basic]

Public Property ReuseAddress As Boolean

[C#]

public bool ReuseAddress {get; set;}

Property Value

Returns **true** if an address can be reused; otherwise returns **false**. The default value is **true**.

Remarks

The **ReuseAddress** property is used to determine if the local address and port number can be reused when starting a new instance of the server. Setting this property to **true** enables a server application to listen for connections using the specified address and port number even if they were in use recently. This is typically used to enable the server to close the listening socket and immediately reopen it without getting an error that the address is in use.

When a listening socket closed, the socket will normally go into a TIME-WAIT state where the local address and port number cannot be immediately reused. A consequence of this is that calling the **Stop** method immediately followed by the **Start** method using the same address and port number values may result in an error indicating that the specified address is already in use. By setting this property to **true**, that error is avoided and the listening socket can be created immediately without waiting for the TIME-WAIT period to elapse. Note that calling the **Restart** method allows the local address and port number to be reused, regardless of this property value.

If you wish to determine if a local port number is already in use by another application, set this property to **false** and attempt to start the server using that port number. If another application is already using that port number, an error will be generated indicating that the address is in use and the server could not be started.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Secure Property

Gets and sets a value which specifies if client connections are secure.

[Visual Basic]

Public Property Secure As Boolean

[C#]

public bool Secure {get; set;}

Property Value

Returns **true** if a secure connections are enabled; otherwise returns **false**. The default value is **false**.

Remarks

The **Secure** property determines if client connections are encrypted using the standard TLS security protocol. The default value for this property is **false**, which specifies that clients will use a standard, unencrypted connection to the server. To enable secure connections, the application should set this property value to **true** prior to calling the **Start** method.

When secure connections are enabled, the server will accept the client connection and then wait for the client to initiate the handshake where both the client and server negotiate the various encryption options available. This process is handled automatically by the server, and all that is required is that the application specify the server certificate which should be used. This is done by setting the **CertificateName** property, and optionally the **CertificateStore** property if required.

It is recommended that the application use exception handling to catch any errors that may occur when changing the value of this property. If the class is unable to initialize the Windows security libraries, an exception will be thrown when this property value is modified.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [CertificateName Property](#) | [CertificateStore Property](#) | [SecureProtocol Property](#)

InternetServer.SecureProtocol Property

Gets and sets a value which specifies the protocol used for secure client connections.

[Visual Basic]

Public Property SecureProtocol As [SecurityProtocols](#)

[C#]

public [InternetServer.SecurityProtocols](#) SecureProtocol {get; set;}

Property Value

A [SecurityProtocols](#) enumeration value which identifies the protocol to be used when accepting a secure client connection.

Remarks

The **SecureProtocol** property can be used to specify the security protocol to be used when accepting a secure connection. By default, the class will attempt to use either TLS 1.2 or TLS 1.3 to accept the connection, with the appropriate protocol automatically selected based on the capabilities of the client. It is recommended that you only change this property value if you fully understand the implications of doing so. Assigning a value to this property will override the default protocol and force the class to attempt to use only the protocol specified.

Attempting to set this property after the server has been started will result in an exception being thrown. This property should only be set after setting the **Secure** property to **true** and before calling the **Start** method.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.ServerAddress Property

Gets and sets the address that will be used by the server to listen for connections.

[Visual Basic]

Public Property ServerAddress As String

[C#]

public string ServerAddress {get; set;}

Property Value

A string which specifies the IP address that the server will use to listen for incoming client connections. An empty string indicates that the server will accept connections on any valid network interface configured for the local system.

Remarks

The **ServerAddress** property is used to specify the default address that the server will use when listening for connections. Setting this property to the value 0.0.0.0 or an empty string indicates that the server should listen for client connections using any valid network interface. If an address is specified, it must be a valid Internet address that is bound to a network adapter configured on the local system. Clients will only be able to connect to the server using that specific address.

It is common to set this property to the value 127.0.0.1 for testing purposes. It is a non-routable address that specifies the local system, and most software firewalls are configured so they do not block applications using this address.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [ServerName Property](#) | [ServerPort Property](#)

InternetServer.ServerName Property

Gets a value which specifies the host name for the local system.

[Visual Basic]

```
Public ReadOnly Property ServerName As String
```

[C#]

```
public string ServerName {get;}
```

Property Value

A string which specifies the local host name.

Remarks

The **ServerName** property returns the fully-qualified host name assigned to the local system. This consists of the local computer name and its domain name. The actual value returned depends on the system configuration. If no domain has been specified for the system, then only the machine name will be returned.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.ServerPort Property

Gets and sets the port number that will be used by the server to listen for connections.

[Visual Basic]

Public Property ServerPort As Integer

[C#]

public int ServerPort {get; set;}

Property Value

An integer value which specifies the port number.

Remarks

The **ServerPort** property is used to set the port number that server will use to listen for incoming client connections. Valid port numbers are in the range of 1 to 65535. It is recommended that most custom servers specify a port number larger than 5000 to avoid potential conflicts with standard Internet services and ephemeral ports used by client applications.

If a port number is specified that is already in use by another application, the **OnError** event will fire and the background server thread will terminate. To enable a server to be stopped and immediately restarted using the same address and port number, make sure that the **ReuseAddress** property is set to a value of **true**.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [ServerAddress Property](#) | [ServerName Property](#)

InternetServer.ServerThread Property

Gets the thread ID for the current server.

[Visual Basic]

```
Public ReadOnly Property ServerThread As Integer
```

[C#]

```
public int ServerThread {get;}
```

Property Value

An integer value which identifies the server thread that was created. A return value of zero specifies that no server has been started.

Remarks

Until the thread terminates, the thread identifier uniquely identifies the thread throughout the system.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Status Property

Gets a value which specifies the current status of the server.

[Visual Basic]

Public **ReadOnly** **Property** **Status** As [ServerStatus](#)

[C#]

public [InternetServer.ServerStatus](#) **Status** {get;}

Property Value

A [ServerStatus](#) enumeration value which specifies the current server status.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.ThrowError Property

Gets and sets a value which specifies if method calls should throw exceptions when an error occurs.

[Visual Basic]

Public Property ThrowError As Boolean

[C#]

public bool ThrowError {get; set;}

Property Value

Returns **true** if method calls will generate exceptions when an error occurs; otherwise returns **false**. The default value is **false**.

Remarks

Error handling for when calling class methods can be done in either of two different styles, according to the value of this property.

If the **ThrowError** property is set to **false**, the application should check the return value of any method that is used, and report errors based upon the documented value of the return code. It is the responsibility of the application to interpret the error code, if it is desired to explain the error in addition to reporting it. This is the default behavior.

If the **ThrowError** property is set to **true**, then exceptions will be generated whenever a method call fails. The program must be written to catch these exceptions and take the appropriate action when an error occurs. Failure to handle an exception will cause the program to terminate abnormally.

Note that if an error occurs while a property is being read or modified, an exception will be raised regardless of the value of the **ThrowError** property.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Timeout Property

Gets and sets a value which specifies a timeout period in seconds.

[Visual Basic]

Public Property Timeout As Integer

[C#]

public int Timeout {get; set;}

Property Value

An integer value which specifies a timeout period in seconds.

Remarks

Setting the **Timeout** property specifies the number of seconds until a socket operation fails and returns an error.

For most server applications it is recommended the timeout period be set between 10 and 20 seconds. It is not recommended that you set the timeout period to zero.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Trace Property

Gets and sets a value which indicates if network function logging is enabled.

[Visual Basic]

```
Public Property Trace As Boolean
```

[C#]

```
public bool Trace {get; set;}
```

Property Value

Returns **true** if network function tracing is enabled; otherwise returns **false**. The default value is **false**.

Remarks

The **Trace** property is used to enable (or disable) the tracing of network function calls. When enabled, each function call is logged to a file, including the function parameters, return value and error code if applicable. This facility can be enabled and disabled at run time, and the trace log file can be specified by setting the **TraceFile** property. All function calls that are being logged are appended to the trace file, if it exists. If no trace file exists when tracing is enabled, the trace file is created.

The tracing facility is available in all of the SocketTools networking classes and is enabled or disabled for an entire process. This means that once trace logging is enabled for a given component, all of the function calls made by the process using any of the SocketTools classes will be logged. For example, if you have an application using both the File Transfer Protocol and Post Office Protocol classes, and you set the **Trace** property to **true**, function calls made by both classes will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all classes used by the process.

If trace logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which multiple processes are being traced or the logfile is fairly large. Since logfiles can grow very quickly, even with modest applications, it is recommended that you delete the file when it is no longer needed.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.TraceFile Property

Gets and sets a value which specifies the name of the network function tracing logfile.

[Visual Basic]

```
Public Property TraceFile As String
```

[C#]

```
public string TraceFile {get; set;}
```

Property Value

A string which specifies the name of the file.

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when network function tracing is enabled. If this property is set to an empty string (the default value), then a file named **SocketTools.log** is created in the system's temporary directory. If no temporary directory exists, then the file is created in the current working directory.

If the file exists, the trace output is appended to the file, otherwise the file is created. Since network function tracing is enabled per-process, the trace file is shared by all instances of the class being used. If multiple class instances have tracing enabled, the **TraceFile** property should be set to the same value for each instance. Since trace files can grow very quickly, even with modest applications, it is recommended that you delete the file when it is no longer needed.

The trace file has the following format:

```
MyApp INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0 MyApp WRN:  
connect(46, 192.0.0.1:1234, 16) returned -1 [10035] MyApp ERR: accept(46, NULL,  
0x0) returned -1 [10038]
```

The first column contains the name of the process that is being traced. The second column identifies if the trace record is reporting information, a warning, or an error. What follows is the name of the function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record (the value is placed inside brackets).

If parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a pointer (a memory address), it is recorded as a hexadecimal value preceded with "0x". A special type of pointer, called a null pointer, is recorded as NULL. Those functions which expect socket addresses are displayed in the following format:

```
aa.bb.cc.dd:nnnn
```

The first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. Note that in the second line of the above example, the control is attempting to connect to a system with the IP address 192.0.0.1 on port 1234.

Note that if the specified file cannot be created, or the user does not have permission to modify an existing file, the error is silently ignored and no trace output will be generated.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.TraceFlags Property

Gets and sets a value which specifies the network function tracing flags.

[Visual Basic]

Public Property TraceFlags As [TraceOptions](#)

[C#]

public [InternetServer.TraceOptions](#) TraceFlags {get; set;}

Property Value

A [TraceOptions](#) enumeration which specifies the amount of detail written to the trace logfile.

Remarks

The **TraceFlags** property is used to specify the type of information written to the trace file when network function tracing is enabled.

Because network function tracing is enabled per-process, the trace flags are shared by all instances of the class being used. If multiple class instances have tracing enabled, the **TraceFlags** property should be set to the same value for each instance. Changing the trace flags for any one instance of the class will affect the logging performed for all SocketTools classes used by the application.

Warnings are generated when a non-fatal error is returned by a network function. For example, if data is being written and the error **OperationWouldBlock** occurs, a warning is generated because the application simply needs to attempt to write the data at a later time.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Version Property

Gets a value which returns the current version of the InternetServer class library.

[Visual Basic]

Public ReadOnly Property Version As String

[C#]

public string Version {get;}

Property Value

A string which specifies the version of the class library.

Remarks

The **Version** property returns a string which identifies the current version and build of the InternetServer class library. This value can be used by an application for validation and debugging purposes.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer Methods

The methods of the **InternetServer** class are listed below. For a complete list of **InternetServer** class members, see the [InternetServer Members](#) topic.

Public Instance Methods

Abort	Overloaded. Abort the connection with a remote host.
Broadcast	Overloaded. Broadcast data to all active clients connected to the server
Cancel	Overloaded. Cancel the current blocking socket operation.
Disconnect	Overloaded. Disconnect the specified client connection from the server.
Dispose	Overloaded. Releases all resources used by InternetServer .
Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
FindClient	Overloaded. Return the socket handle for the client session with the specified moniker.
Flush	Overloaded. Flush the contents of the send and receive buffers for the client socket.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
Initialize	Overloaded. Initialize an instance of the InternetServer class.
Lock	Lock the server to synchronize access to shared data for all active client sessions.
Peek	Overloaded. Read data from the client and store it in a byte array, but do not remove the data from the socket buffers.
Read	Overloaded. Read data from the client socket and store it in a byte array.
ReadLine	Overloaded. Read up to a line of data from the client and return it in a string buffer.
Reject	Overloaded. Rejects a connection request from a client.
Reset	Reset the internal state of the object, resetting all properties to their default values.
Resolve	Resolves a host name to a host IP address.

Restart	Restarts the server and terminates all active client connections.
Resume	Resume accepting new client connections.
Start	Overloaded. Start listening for client connections on the specified IP address and port number.
Stop	Stop listening for new client connections and terminate all active clients already connected to the server.
Suspend	Overloaded. Suspend accepting new client connections with additional options.
Throttle	Overloaded. Limit the maximum number of client connections, connections per IP address and connection rate.
ToString (inherited from Object)	Returns a String that represents the current Object.
Uninitialize	Uninitialize the class library and release any resources allocated for the server.
Unlock	Unlock the server and allow other server threads to resume execution.
Write	Overloaded. Write one or more bytes of data to a client.
WriteLine	Overloaded. Send a line of text to a client, terminated by a carriage-return and linefeed.

Protected Instance Methods

 Dispose	Overloaded. Releases the unmanaged resources allocated by the InternetServer class and optionally releases the managed resources.
 Finalize	Destroys an instance of the class, releasing the resources allocated for the session and unloading the networking library.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Abort Method

Abort the connection with a remote host.

Overload List

Abort the connection with a remote host.

```
public void Abort();
```

Abort the connection with a remote host.

```
public void Abort(int);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [Disconnect Method](#)

InternetServer.Abort Method ()

Abort the connection with a remote host.

[Visual Basic]

```
Overloads Public Sub Abort()
```

[C#]

```
public void Abort();
```

Remarks

The **Abort** method immediately terminates the client connection, without waiting for any remaining data in the socket buffer to be written out. This method should only be used when the connection must be closed immediately. If this method is used, the client will see the connection as being terminated abnormally.

It is recommended that applications using the **Disconnect** method unless it is absolutely necessary to terminate the connection and immediately release the socket handle.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Abort** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Abort Overload List](#) | [Disconnect Method](#)

InternetServer.Abort Method (Int32)

Abort the connection with a remote host.

[Visual Basic]

```
Overloads Public Sub Abort( _  
    ByVal handle As Integer _  
)
```

[C#]

```
public void Abort(  
    int handle  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

Remarks

The **Abort** method immediately terminates the client connection, without waiting for any remaining data in the socket buffers to be written out. This method should only be used when the connection must be closed immediately. If this method is used, the client will see the connection as being terminated abnormally.

It is recommended that applications using the **Disconnect** method unless it is absolutely necessary to terminate the connection and immediately release the socket handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Abort Overload List](#) | [Disconnect Method](#)

InternetServer.Broadcast Method

Broadcast data to all active clients connected to the server

Overload List

Broadcast data to all active clients connected to the server

```
public int Broadcast(byte[]);
```

Broadcast data to all active clients connected to the server

```
public int Broadcast(byte[],int);
```

Broadcast data to all active clients connected to the server

```
public int Broadcast(string);
```

Broadcast data to all active clients connected to the server

```
public int Broadcast(string,int);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Broadcast Method (Byte[])

Broadcast data to all active clients connected to the server

[Visual Basic]

```
Overloads Public Function Broadcast( _  
    ByVal buffer As Byte() _  
) As Integer
```

[C#]

```
public int Broadcast(  
    byte[] buffer  
);
```

Parameters

buffer

A byte array that contains the data that will be broadcast.

Return Value

An integer value which specifies the number of clients that the data was broadcast to. A return value of -1 indicates an error condition, and the value of the **LastError** property will indicate the cause of the failure.

Remarks

The **Broadcast** method broadcasts contents of the specified byte array to all clients connected to the server. If this method is called inside a server event handler, the message is broadcast to all clients except for the current, active client that is processing the event notification. If this method is called outside of an event handler, the data is broadcast to all connected clients.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Broadcast Overload List](#)

InternetServer.Broadcast Method (Byte[], Int32)

Broadcast data to all active clients connected to the server

[Visual Basic]

```
Overloads Public Function Broadcast( _  
    ByVal buffer As Byte(), _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Broadcast(  
    byte[] buffer,  
    int length  
);
```

Parameters

buffer

A byte array that contains the data that will be broadcast.

length

An integer value which specifies the maximum number of bytes of data to broadcast. This value cannot be larger than the size of the buffer specified by the caller.

Return Value

An integer value which specifies the number of clients that the data was broadcast to. A return value of -1 indicates an error condition, and the value of the **LastError** property will indicate the cause of the failure.

Remarks

The **Broadcast** method broadcasts contents of the specified byte array to all clients connected to the server. If this method is called inside a server event handler, the message is broadcast to all clients except for the current, active client that is processing the event notification. If this method is called outside of an event handler, the data is broadcast to all connected clients.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Broadcast Overload List](#)

InternetServer.Broadcast Method (String)

Broadcast data to all active clients connected to the server

[Visual Basic]

```
Overloads Public Function Broadcast( _  
    ByVal buffer As String _  
) As Integer
```

[C#]

```
public int Broadcast(  
    string buffer  
);
```

Parameters

buffer

A string that contains the data that will be broadcast.

Return Value

An integer value which specifies the number of clients that the data was broadcast to. A return value of -1 indicates an error condition, and the value of the **LastError** property will indicate the cause of the failure.

Remarks

The **Broadcast** method broadcasts contents of the specified string to all clients connected to the server. If this method is called inside a server event handler, the message is broadcast to all clients except for the current, active client that is processing the event notification. If this method is called outside of an event handler, the data is broadcast to all connected clients.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Broadcast Overload List](#)

InternetServer.Broadcast Method (String, Int32)

Broadcast data to all active clients connected to the server

[Visual Basic]

```
Overloads Public Function Broadcast( _  
    ByVal buffer As String, _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Broadcast(  
    string buffer,  
    int length  
);
```

Parameters

buffer

A string that contains the data that will be broadcast.

length

An integer value which specifies the maximum number of bytes of data to broadcast. This value cannot be larger than the size of the buffer specified by the caller.

Return Value

An integer value which specifies the number of clients that the data was broadcast to. A return value of -1 indicates an error condition, and the value of the **LastError** property will indicate the cause of the failure.

Remarks

The **Broadcast** method broadcasts contents of the specified string to all clients connected to the server. If this method is called inside a server event handler, the message is broadcast to all clients except for the current, active client that is processing the event notification. If this method is called outside of an event handler, the data is broadcast to all connected clients.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Broadcast Overload List](#)

InternetServer.Cancel Method

Cancel the current blocking socket operation.

Overload List

Cancel the current blocking socket operation.

```
public void Cancel();
```

Cancel the current blocking socket operation.

```
public void Cancel(int);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Cancel Method ()

Cancel the current blocking socket operation.

[Visual Basic]

Overloads Public Sub Cancel()

[C#]

public void Cancel();

Remarks

When the **Cancel** method is called, the blocking socket operation will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other blocking function. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Cancel** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Cancel Overload List](#)

InternetServer.Cancel Method (Int32)

Cancel the current blocking socket operation.

[Visual Basic]

```
Overloads Public Sub Cancel( _  
    ByVal handle As Integer _  
)
```

[C#]

```
public void Cancel(  
    int handle  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

Remarks

When the **Cancel** method is called, the blocking socket operation will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other blocking function. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Cancel Overload List](#)

InternetServer.Disconnect Method

Disconnect the specified client connection from the server.

Overload List

Disconnect the specified client connection from the server.

```
public void Disconnect();
```

Disconnect the specified client connection from the server.

```
public void Disconnect(int);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Disconnect Method ()

Disconnect the specified client connection from the server.

[Visual Basic]

Overloads Public Sub Disconnect()

[C#]

public void Disconnect();

Remarks

The **Disconnect** method terminates the active client connection with the server and closes the socket handle allocated by the class. Note that the client socket is not immediately released when the connection is terminated and will enter a wait state for two minutes. After the time wait period has elapsed, the socket will be released by the operating system. This is a normal safety mechanism to handle any packets that may arrive after the connection has been closed.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Disconnect** method outside of an event handler, you must explicitly specify the client handle.

To immediately terminate the connection and release the socket, use the **Abort** method.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Disconnect Overload List](#)

InternetServer.Disconnect Method (Int32)

Disconnect the specified client connection from the server.

[Visual Basic]

```
Overloads Public Sub Disconnect( _  
    ByVal handle As Integer _  
)
```

[C#]

```
public void Disconnect(  
    int handle  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

Remarks

The **Disconnect** method terminates the specified client connection with the server and closes the socket handle allocated by the class. Note that the client socket is not immediately released when the connection is terminated and will enter a wait state for two minutes. After the time wait period has elapsed, the socket will be released by the operating system. This is a normal safety mechanism to handle any packets that may arrive after the connection has been closed.

To immediately terminate the connection and release the socket, use the **Abort** method.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Disconnect Overload List](#) | [Abort Method](#)

InternetServer.Dispose Method

Releases all resources used by [InternetServer](#).

Overload List

Releases all resources used by [InternetServer](#).

```
public void Dispose();
```

Releases the unmanaged resources allocated by the [InternetServer](#) class and optionally releases the managed resources.

```
protected void Dispose(bool);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Dispose Method ()

Releases all resources used by [InternetServer](#).

[Visual Basic]

```
NotOverridable Overloads Public Sub Dispose() _  
    Implements IDisposable.Dispose
```

[C#]

```
public void Dispose();
```

Implements

IDisposable.Dispose

Remarks

The **Dispose** method stops the server, terminates all active client sessions and explicitly releases the resources allocated for this instance of the class. In some cases, better performance can be achieved if the programmer explicitly releases resources when they are no longer being used. The **Dispose** method provides explicit control over these resources.

Unlike the **Uninitialize** method, once the **Dispose** method has been called, that instance of the class cannot be re-initialized and you should not attempt to access class properties or invoke any methods. Note that this method can be called even if other references to the object are active.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Dispose Overload List](#)

InternetServer.Dispose Method (Boolean)

Releases the unmanaged resources allocated by the [InternetServer](#) class and optionally releases the managed resources.

[Visual Basic]

```
Overloads Protected Sub Dispose( _  
    ByVal disposing As Boolean _  
)
```

[C#]

```
protected void Dispose(  
    bool disposing  
);
```

Parameters

disposing

A boolean value which should be specified as **true** to release both managed and unmanaged resources; **false** to release only unmanaged resources.

Remarks

The **Dispose** method terminates any active connection and explicitly releases the resources allocated for this instance of the class. In some cases, better performance can be achieved if the programmer explicitly releases resources when they are no longer being used. The **Dispose** method provides explicit control over these resources.

Unlike the **Uninitialize** method, once the **Dispose** method has been called, that instance of the class cannot be re-initialized and you should not attempt to access class properties or invoke any methods. Note that this method can be called even if other references to the object are active.

You should call **Dispose** in your derived class when you are finished using the derived class. The **Dispose** method leaves the derived class in an unusable state. After calling **Dispose**, you must release all references to the derived class and the **InternetServer** class so the memory they were occupying can be reclaimed by garbage collection.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Dispose Overload List](#)

InternetServer.Finalize Method

Destroys an instance of the class, releasing the resources allocated for the session and unloading the networking library.

[Visual Basic]

Overrides Protected Sub Finalize()

[C#]

protected override void Finalize();

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.FindClient Method

Return the socket handle for the client session with the specified ID.

Overload List

Return the socket handle for the client session with the specified ID.

```
public int FindClient(int);
```

Return the socket handle for the client session with the specified moniker.

```
public int FindClient(string);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [ClientId Property](#)

InternetServer.FindClient Method (Int32)

Return the socket handle for the client session with the specified ID.

[Visual Basic]

```
Overloads Public Function FindClient( _  
    ByVal clientId As Integer _  
) As Integer
```

[C#]

```
public int FindClient(  
    int clientId  
);
```

Parameters

clientId

An integer value which uniquely identifies the client session.

Return Value

An integer value which specifies the socket handle for the client session. If the specified client ID does not match an active client session, the method will return a value of -1 and the value of the **LastError** property will indicate the cause of the failure.

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value. This value can be used by the application to identify that client session, and is different than the socket handle allocated for the client. While it is possible for a client socket handle to be reused by the operating system, client IDs are unique throughout the life of the server session and are never duplicated.

The application can determine the ID assigned to the current client session using the **ClientId** property from within an event handler such as **OnConnect**.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.FindClient Overload List](#) | [ClientId Property](#)

InternetServer.FindClient Method (String)

Return the socket handle for the client session with the specified moniker.

[Visual Basic]

```
Overloads Public Function FindClient( _  
    ByVal clientName As String _  
) As Integer
```

[C#]

```
public int FindClient(  
    string clientName  
);
```

Parameters

clientName

A string which specifies the moniker for the client session.

Return Value

An integer value which specifies the socket handle for the client session. If the specified moniker does not match an active client session, the method will return a value of -1 and the value of the **LastError** property will indicate the cause of the failure.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session by setting the **ClientName** property from within a class event handler such as the **OnConnect** event.

Monikers are not case-sensitive, and they must be unique so that no client socket for a particular server can have the same moniker. The maximum length for a moniker is 127 characters.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.FindClient Overload List](#) | [ClientName Property](#)

InternetServer.Flush Method

Flush the contents of the send and receive buffers for the client socket.

Overload List

Flush the contents of the send and receive buffers for the client socket.

```
public bool Flush();
```

Flush the contents of the send and receive buffers for the client socket.

```
public bool Flush(int);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [Abort Method](#) | [Disconnect Method](#) | [Flush Method](#)

InternetServer.Flush Method ()

Flush the contents of the send and receive buffers for the client socket.

[Visual Basic]

Overloads Public Function Flush() As Boolean

[C#]

public bool Flush();

Return Value

A boolean value which specifies if the client buffers have been flushed.

Remarks

The **Flush** method will flush any data waiting to be read or written for the active client session. It is important to note that this method is not similar to flushing data to a disk file; it does not ensure that a specific block of data has been written to the client. For example, you should never call this function immediately after calling the **Write** or **WriteLine** methods, or prior to calling the **Disconnect** method.

An server should never use the **Flush** method under normal circumstances. This method is only to be used when the server needs to immediately return the client to an inactive state with no pending data to be read or written. Calling this method may result in data loss and should only be used if you understand the implications of discarding any data which has been sent by the client.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Flush** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Flush Overload List](#) | [Abort Method](#) | [Disconnect Method](#) | [Flush Method](#)

InternetServer.Flush Method (Int32)

Flush the contents of the send and receive buffers for the client socket.

[Visual Basic]

```
Overloads Public Function Flush( _  
    ByVal handle As Integer _  
) As Boolean
```

[C#]

```
public bool Flush(  
    int handle  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

Return Value

A boolean value which specifies if the client buffers have been flushed.

Remarks

The **Flush** method will flush any data waiting to be read or written for the specified client session. It is important to note that this method is not similar to flushing data to a disk file; it does not ensure that a specific block of data has been written to the client. For example, you should never call this function immediately after calling the **Write** or **WriteLine** methods, or prior to calling the **Disconnect** method.

An server should never use the **Flush** method under normal circumstances. This method is only to be used when the server needs to immediately return the client to an inactive state with no pending data to be read or written. Calling this method may result in data loss and should only be used if you understand the implications of discarding any data which has been sent by the client.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Flush Overload List](#) | [Abort Method](#) | [Disconnect Method](#) | [Flush Method](#)

InternetServer.Initialize Method

Initialize an instance of the InternetServer class.

Overload List

Initialize an instance of the InternetServer class.

```
public bool Initialize();
```

Initialize an instance of the InternetServer class.

```
public bool Initialize(string);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [Uninitialize Method](#)

InternetServer.Initialize Method ()

Initialize an instance of the InternetServer class.

[Visual Basic]

```
Overloads Public Function Initialize() As Boolean
```

[C#]

```
public bool Initialize();
```

Return Value

A boolean value which specifies if the class was initialized successfully.

Remarks

The Initialize method can be used to explicitly initialize an instance of the InternetServer class, loading the networking library and allocating resources for the current thread. Typically it is not necessary to explicitly call this method because the instance of the class is initialized by the class constructor. However, if the **Uninitialize** method is called, the class must be re-initialized before any other methods are called.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Initialize Overload List](#) | [Uninitialize Method](#)

InternetServer.Initialize Method (String)

Initialize an instance of the InternetServer class.

[Visual Basic]

```
Overloads Public Function Initialize( _  
    ByVal LicenseKey As String _  
) As Boolean
```

[C#]

```
public bool Initialize(  
    string LicenseKey  
);
```

Return Value

A boolean value which specifies if the class was initialized successfully.

Remarks

The Initialize method can be used to explicitly initialize an instance of the InternetServer class, loading the networking library and allocating resources for the current thread. Typically an application would define the license key as a custom attribute, however this method can be used to initialize the class directly.

The runtime license key for your copy of InternetServer can be generated using the License Manager utility that is included with the product. Note that if you have installed an evaluation license, you will not have a runtime license key and cannot redistribute any applications which use the InternetServer class.

Example

The following example shows how to use the Initialize method to initialize an instance of the class. This example assumes that the license key string has been defined in code.

```
SocketTools.InternetServer server = new SocketTools.InternetServer();  
  
if (server.Initialize(strLicenseKey) == false)  
{  
    MessageBox.Show(server.LastErrorString, "Error",  
        MessageBoxButtons.OK, MessageBoxIcon.Exclamation);  
    return;  
}  
  
Dim Server As New SocketTools.InternetServer  
  
If Server.Initialize(strLicenseKey) = False Then  
    MsgBox(Server.LastErrorString, vbIconExclamation)  
    Exit Sub  
End If
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Initialize Overload List](#) | [RuntimeLicenseAttribute Class](#) | [Uninitialize Method](#)

InternetServer.Lock Method

Lock the server to synchronize access to shared data for all active client sessions.

[Visual Basic]

```
Public Function Lock() As Boolean
```

[C#]

```
public bool Lock();
```

Return Value

A boolean value which specifies if the server was locked. A return value of **true** specifies that the server was locked, and all other threads being managed by the server have been blocked. A return value of **false** indicates that the server could not be locked, typically because a potential deadlock was detected.

Remarks

The **Lock** method causes the server to enter a locked state where only the current thread may interact with the server and the clients that are connected to it. While a server is locked, all other threads will block when they attempt to perform a network operation. When the server is unlocked, the blocked threads will resume normal execution.

This method should be used carefully, and a server should never be left in a locked state for an extended period of time. It is meant to be used when the server process updates a global data structure and it must prevent any other threads from performing a network operation during the update. Only one server can be locked at any one time, and once a server has been locked, it can only be unlocked by the same thread.

The program should always check the return value from this method, and should never assume that the lock has been established. If more than one thread attempts to lock a server at the same time, there is no guarantee as to which thread will actually establish the lock. If a potential deadlock situation is detected, this function will fail and return a value of false.

Every time the **Lock** method is called, an internal lock counter is incremented, and the lock will not be released until the lock count drops to zero. This means that each call to the **Lock** method must be matched by an equal number of calls to the **Unlock** method. Failure to do so will result in the server becoming non-responsive as it remains in a locked state.

The **IsLocked** property can be used to determine if the server has been locked.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [Unlock Method](#) | [IsLocked Property](#)

InternetServer.Peek Method

Return the number of bytes available to be read from the client socket.

Overload List

Return the number of bytes available to be read from the client socket.

```
public int Peek();
```

Read data from the client and store it in a byte array, but do not remove the data from the socket buffers.

```
public int Peek(byte[]);
```

Read data from the client and store it in a byte array, but do not remove the data from the socket buffers.

```
public int Peek(byte[],int);
```

Return the number of bytes available to be read from the client socket.

```
public int Peek(int);
```

Read data from the client and store it in a byte array, but do not remove the data from the socket buffers.

```
public int Peek(int,byte[]);
```

Read data from the client and store it in a byte array, but do not remove the data from the socket buffers.

```
public int Peek(int,byte[],int);
```

Read data from the client and store it in a string, but do not remove the data from the socket buffers.

```
public int Peek(int,ref string);
```

Read data from the client and store it in a string, but do not remove the data from the socket buffers.

```
public int Peek(int,ref string,int);
```

Read data from the client and store it in a string, but do not remove the data from the socket buffers.

```
public int Peek(ref string);
```

Read data from the client and store it in a string, but do not remove the data from the socket buffers.

```
public int Peek(ref string,int);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [IsReadable Property](#) | [OnRead Event](#)

InternetServer.Peek Method ()

Return the number of bytes available to be read from the client socket.

[Visual Basic]

Overloads Public Function Peek() As Integer

[C#]

public int Peek();

Return Value

An integer value which specifies the number of bytes available to be read from the client socket. A return value of zero specifies that there is no data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Peek** method returns the number of bytes that can be read in a single operation. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. Using this method in a loop to poll a socket can cause the application to become non-responsive. To determine if there is data available to be read, use the **IsReadable** property.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Peek** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Peek Overload List](#) | [IsReadable Property](#) | [OnRead Event](#)

InternetServer.Peek Method (Byte[])

Read data from the client and store it in a byte array, but do not remove the data from the socket buffers.

[Visual Basic]

```
Overloads Public Function Peek( _  
    ByVal buffer As Byte() _  
) As Integer
```

[C#]

```
public int Peek(  
    byte[] buffer  
);
```

Parameters

buffer

A byte array that the data will be stored in.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that there is no data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Peek** method returns data that is available to read from the client. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. To determine if there is data available to be read, use the **IsReadable** property.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Peek** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Peek Overload List](#) | [Read Method](#) | [IsReadable Property](#) | [OnRead Event](#)

InternetServer.Peek Method (Byte[], Int32)

Read data from the client and store it in a byte array, but do not remove the data from the socket buffers.

[Visual Basic]

```
Overloads Public Function Peek( _  
    ByVal buffer As Byte(), _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Peek(  
    byte[] buffer,  
    int length  
);
```

Parameters

buffer

A byte array that the data will be stored in.

length

An integer value which specifies the maximum number of bytes of data to read. This value cannot be larger than the size of the buffer specified by the caller.

Return Value

An integer value which specifies the number of bytes actually read from the client. A return value of zero specifies that there is no data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Peek** method returns data that is available to read from the socket, up to the number of bytes specified. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. To determine if there is data available to be read, use the **IsReadable** property.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Peek** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Peek Overload List](#) | [Read Method](#) | [IsReadable Property](#) | [OnRead Event](#)

InternetServer.Peek Method (Int32)

Return the number of bytes available to be read from the client socket.

[Visual Basic]

```
Overloads Public Function Peek( _  
    ByVal handle As Integer _  
) As Integer
```

[C#]

```
public int Peek(  
    int handle  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

Return Value

An integer value which specifies the number of bytes available to be read from the specified client socket. A return value of zero specifies that there is no data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Peek** method returns the number of bytes that can be read in a single operation. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. Using this method in a loop to poll a socket can cause the application to become non-responsive. To determine if there is data available to be read, use the **IsReadable** property.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Peek Overload List](#) | [OnRead Event](#)

InternetServer.Peek Method (Int32, Byte[])

Read data from the client and store it in a byte array, but do not remove the data from the socket buffers.

[Visual Basic]

```
Overloads Public Function Peek( _  
    ByVal handle As Integer, _  
    ByVal buffer As Byte() _  
) As Integer
```

[C#]

```
public int Peek(  
    int handle,  
    byte[] buffer  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A byte array that the data will be stored in.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that there is no data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Peek** method returns data that is available to read from the client, up to the number of bytes specified. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Peek Overload List](#) | [Read Method](#) | [OnRead Event](#)

InternetServer.Peek Method (Int32, Byte[], Int32)

Read data from the client and store it in a byte array, but do not remove the data from the socket buffers.

[Visual Basic]

```
Overloads Public Function Peek( _  
    ByVal handle As Integer, _  
    ByVal buffer As Byte(), _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Peek(  
    int handle,  
    byte[] buffer,  
    int length  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A byte array that the data will be stored in.

length

An integer value which specifies the maximum number of bytes of data to read. This value cannot be larger than the size of the buffer specified by the caller.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that there is no data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Peek** method returns data that is available to read from the client, up to the number of bytes specified. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Peek Overload List](#) | [Read Method](#) | [OnRead Event](#)

InternetServer.Peek Method (Int32, String)

Read data from the client and store it in a string, but do not remove the data from the socket buffers.

[Visual Basic]

```
Overloads Public Function Peek( _  
    ByVal handle As Integer, _  
    ByRef buffer As String _  
) As Integer
```

[C#]

```
public int Peek(  
    int handle,  
    ref string buffer  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A string that will contain the data read from the socket.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that there is no data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Peek** method returns data that is available to read from the socket, up to a maximum of 8192 bytes. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

This method should only be used if the remote host is sending data that consists of printable characters. Binary data should be read using the method that accepts a byte array as the buffer parameter.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Peek Overload List](#) | [Read Method](#) | [OnRead Event](#)

InternetServer.Peek Method (Int32, String, Int32)

Read data from the client and store it in a string, but do not remove the data from the socket buffers.

[Visual Basic]

```
Overloads Public Function Peek( _  
    ByVal handle As Integer, _  
    ByRef buffer As String, _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Peek(  
    int handle,  
    ref string buffer,  
    int length  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A string that will contain the data read from the client.

length

An integer value which specifies the maximum number of bytes of data to read.

Return Value

An integer value which specifies the number of bytes actually read from the client. A return value of zero specifies that there is no data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Peek** method returns data that is available to read from the client, up to the number of bytes specified. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

This method should only be used if the remote host is sending data that consists of printable characters. Binary data should be read using the method that accepts a byte array as the buffer parameter.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Peek Overload List](#) | [Read Method](#) | [OnRead Event](#)

InternetServer.Peek Method (String)

Read data from the client and store it in a string, but do not remove the data from the socket buffers.

[Visual Basic]

```
Overloads Public Function Peek( _  
    ByRef buffer As String _  
) As Integer
```

[C#]

```
public int Peek(  
    ref string buffer  
);
```

Parameters

buffer

A string that will contain the data read from the socket.

Return Value

An integer value which specifies the number of bytes actually read from the client. A return value of zero specifies that there is no data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Peek** method returns data that is available to read from the client, up to a maximum of 8192 bytes. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. To determine if there is data available to be read, use the **IsReadable** property.

This method should only be used if the remote host is sending data that consists of printable characters. Binary data should be read using the method that accepts a byte array as the buffer parameter.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Peek** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Peek Overload List](#) | [Read Method](#) | [IsReadable Property](#) | [OnRead Event](#)

InternetServer.Peek Method (String, Int32)

Read data from the client and store it in a string, but do not remove the data from the socket buffers.

[Visual Basic]

```
Overloads Public Function Peek( _  
    ByRef buffer As String, _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Peek(  
    ref string buffer,  
    int length  
);
```

Parameters

buffer

A string that will contain the data read from the client.

length

An integer value which specifies the maximum number of bytes of data to read.

Return Value

An integer value which specifies the number of bytes actually read from the client. A return value of zero specifies that there is no data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Peek** method returns data that is available to read from the socket, up to the number of bytes specified. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. To determine if there is data available to be read, use the **IsReadable** property.

This method should only be used if the remote host is sending data that consists of printable characters. Binary data should be read using the method that accepts a byte array as the buffer parameter.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Peek** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Peek Overload List](#) | [Read Method](#) | [IsReadable Property](#) | [OnRead Event](#)

InternetServer.Read Method

Read data from the client socket and store it in a byte array.

Overload List

Read data from the client socket and store it in a byte array.

```
public int Read(byte[]);
```

Read data from the client socket and store it in a byte array.

```
public int Read(byte[],int);
```

Read data from the client socket and store it in a byte array.

```
public int Read(int,byte[]);
```

Read data from the client socket and store it in a byte array.

```
public int Read(int,byte[],int);
```

Read data from the client socket and store it in a string.

```
public int Read(int,ref string);
```

Read data from the client socket and store it in a string.

```
public int Read(int,ref string,int);
```

Read data from the client socket and store it in a string.

```
public int Read(ref string);
```

Read data from the client socket and store it in a string.

```
public int Read(ref string,int);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Read Method (Byte[])

Read data from the client socket and store it in a byte array.

[Visual Basic]

```
Overloads Public Function Read( _  
    ByVal buffer As Byte() _  
) As Integer
```

[C#]

```
public int Read(  
    byte[] buffer  
);
```

Parameters

buffer

A byte array that the data will be stored in.

Return Value

An integer value which specifies the number of bytes actually read from the client. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Read** method returns data that has been read from the socket, up to the size of the byte array passed to the method. If no data is available to be read, the calling thread will block until data is received from the server or the connection is closed.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Read** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Read Overload List](#)

InternetServer.Read Method (Byte[], Int32)

Read data from the client socket and store it in a byte array.

[Visual Basic]

```
Overloads Public Function Read( _  
    ByVal buffer As Byte(), _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Read(  
    byte[] buffer,  
    int length  
);
```

Parameters

buffer

A byte array that the data will be stored in.

length

An integer value which specifies the maximum number of bytes of data to read. This value cannot be larger than the size of the buffer specified by the caller.

Return Value

An integer value which specifies the number of bytes actually read from the client. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Read** method returns data that has been read from the socket, up to the number of bytes specified. If no data is available to be read, the calling thread will block until data is received from the server or the connection is closed.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Read** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Read Overload List](#)

InternetServer.Read Method (Int32, Byte[])

Read data from the client socket and store it in a byte array.

[Visual Basic]

```
Overloads Public Function Read( _  
    ByVal handle As Integer, _  
    ByVal buffer As Byte() _  
) As Integer
```

[C#]

```
public int Read(  
    int handle,  
    byte[] buffer  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A byte array that the data will be stored in.

Return Value

An integer value which specifies the number of bytes actually read from the specified client socket. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Read** method returns data that has been read from the socket, up to the size of the byte array passed to the method. If no data is available to be read, the calling thread will block until data is received from the server or the connection is closed.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Read Overload List](#)

InternetServer.Read Method (Int32, Byte[], Int32)

Read data from the client socket and store it in a byte array.

[Visual Basic]

```
Overloads Public Function Read( _  
    ByVal handle As Integer, _  
    ByVal buffer As Byte(), _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Read(  
    int handle,  
    byte[] buffer,  
    int length  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A byte array that the data will be stored in.

length

An integer value which specifies the maximum number of bytes of data to read. This value cannot be larger than the size of the buffer specified by the caller.

Return Value

An integer value which specifies the number of bytes actually read from the specified client socket. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Read** method returns data that has been read from the socket, up to the number of bytes specified. If no data is available to be read, the calling thread will block until data is received from the server or the connection is closed.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Read Overload List](#)

InternetServer.Read Method (Int32, String)

Read data from the client socket and store it in a string.

[Visual Basic]

```
Overloads Public Function Read( _  
    ByVal handle As Integer, _  
    ByRef buffer As String _  
) As Integer
```

[C#]

```
public int Read(  
    int handle,  
    ref string buffer  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A string that will contain the data read from the socket.

Return Value

An integer value which specifies the number of bytes actually read from the specified client socket. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Read** method returns data that has been read from the socket, up to a maximum of 8192 bytes. If no data is available to be read, the calling thread will block until data is received from the server or the connection is closed.

This method should only be used if the remote host is sending data that consists of printable characters. Binary data should be read using the method that accepts a byte array as the buffer parameter.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Read Overload List](#)

InternetServer.Read Method (Int32, String, Int32)

Read data from the client socket and store it in a string.

[Visual Basic]

```
Overloads Public Function Read( _  
    ByVal handle As Integer, _  
    ByRef buffer As String, _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Read(  
    int handle,  
    ref string buffer,  
    int length  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A string that will contain the data read from the socket.

length

An integer value which specifies the maximum number of bytes of data to read.

Return Value

An integer value which specifies the number of bytes actually read from the specified client socket. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Read** method returns data that has been read from the specified client socket, up to the number of bytes specified. If no data is available to be read, the calling thread will block until data is received from the server or the connection is closed.

This method should only be used if the remote host is sending data that consists of printable characters. Binary data should be read using the method that accepts a byte array as the buffer parameter.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Read Overload List](#)

InternetServer.Read Method (String)

Read data from the client socket and store it in a string.

[Visual Basic]

```
Overloads Public Function Read( _  
    ByRef buffer As String _  
) As Integer
```

[C#]

```
public int Read(  
    ref string buffer  
);
```

Parameters

buffer

A string that will contain the data read from the client.

Return Value

An integer value which specifies the number of bytes actually read from the client. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Read** method returns data that has been read from the client, up to a maximum of 8192 bytes. If no data is available to be read, the calling thread will block until data is received from the server or the connection is closed.

This method should only be used if the remote host is sending data that consists of printable characters. Binary data should be read using the method that accepts a byte array as the buffer parameter.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Read** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Read Overload List](#)

InternetServer.Read Method (String, Int32)

Read data from the client socket and store it in a string.

[Visual Basic]

```
Overloads Public Function Read( _  
    ByRef buffer As String, _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Read(  
    ref string buffer,  
    int length  
);
```

Parameters

buffer

A string that will contain the data read from the client.

length

An integer value which specifies the maximum number of bytes of data to read.

Return Value

An integer value which specifies the number of bytes actually read from the client. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Read** method returns data that has been read from the client, up to the number of bytes specified. If no data is available to be read, the calling thread will block until data is received from the server or the connection is closed.

This method should only be used if the remote host is sending data that consists of printable characters. Binary data should be read using the method that accepts a byte array as the buffer parameter.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Read** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Read Overload List](#)

InternetServer.ReadLine Method

Read up to a line of data from the client and return it in a string buffer.

Overload List

Read up to a line of data from the client and return it in a string buffer.

```
public bool ReadLine(int,ref string);
```

Read up to a line of data from the client and return it in a string buffer.

```
public bool ReadLine(int,ref string,int);
```

Read up to a line of data from the client and return it in a string buffer.

```
public bool ReadLine(ref string);
```

Read up to a line of data from the client and return it in a string buffer.

```
public bool ReadLine(ref string,int);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.ReadLine Method (Int32, String)

Read up to a line of data from the client and return it in a string buffer.

[Visual Basic]

```
Overloads Public Function ReadLine( _  
    ByVal handle As Integer, _  
    ByRef buffer As String _  
) As Boolean
```

[C#]

```
public bool ReadLine(  
    int handle,  
    ref string buffer  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A string which will contain the data read from the client.

Return Value

This method returns a boolean value which specifies if a line of data has been read. A value of **true** indicates a line of data has been read. If an error occurs or there is no more data available to read, then the method will return **false**. It is possible for data to be returned in the string buffer even if the return value is **false**.

Applications should check the length of the string after the method returns to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Remarks

The **ReadLine** method reads data from the client until an end-of-line character sequence is encountered. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a single line of text data in a string variable. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point; the string will not contain the carriage-return or linefeed characters.

There are some limitations when using the **ReadLine** method. The method should only be used to read text, never binary data. In particular, it will discard nulls, linefeed and carriage return control characters. This method will force the calling thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection.

The **Read** and **ReadLine** methods can be intermixed, however be aware that the **Read** method will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.ReadLine Overload List](#)

InternetServer.ReadLine Method (Int32, String, Int32)

Read up to a line of data from the client and return it in a string buffer.

[Visual Basic]

```
Overloads Public Function ReadLine( _  
    ByVal handle As Integer, _  
    ByRef buffer As String, _  
    ByVal length As Integer _  
) As Boolean
```

[C#]

```
public bool ReadLine(  
    int handle,  
    ref string buffer,  
    int length  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A string which will contain the data read from the client.

length

An integer value which specifies the maximum number of bytes of data to read.

Return Value

This method returns a boolean value which specifies if a line of data has been read. A value of **true** indicates a line of data has been read. If an error occurs or there is no more data available to read, then the method will return **false**. It is possible for data to be returned in the string buffer even if the return value is **false**.

Applications should check the length of the string after the method returns to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Remarks

The **ReadLine** method reads data from the client up to the specified number of bytes or until an end-of-line character sequence is encountered. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a single line of text data in a string variable. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point; the string will not contain the carriage-return or linefeed characters.

There are some limitations when using the **ReadLine** method. The method should only be used to read text, never binary data. In particular, it will discard nulls, linefeed and carriage return control characters. This method will force the calling thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection.

The **Read** and **ReadLine** methods can be intermixed, however be aware that the **Read** method will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

See Also

InternetServer.ReadLine Method (String)

Read up to a line of data from the client and return it in a string buffer.

[Visual Basic]

```
Overloads Public Function ReadLine( _  
    ByRef buffer As String _  
) As Boolean
```

[C#]

```
public bool ReadLine(  
    ref string buffer  
);
```

Parameters

buffer

A string which will contain the data read from the client.

Return Value

This method returns a boolean value which specifies if a line of data has been read. A value of **true** indicates a line of data has been read. If an error occurs or there is no more data available to read, then the method will return **false**. It is possible for data to be returned in the string buffer even if the return value is **false**.

Applications should check the length of the string after the method returns to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Remarks

The **ReadLine** method reads data from the client until an end-of-line character sequence is encountered. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a single line of text data in a string variable. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point; the string will not contain the carriage-return or linefeed characters.

There are some limitations when using the **ReadLine** method. The method should only be used to read text, never binary data. In particular, it will discard nulls, linefeed and carriage return control characters. This method will force the calling thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection.

The **Read** and **ReadLine** methods can be intermixed, however be aware that the **Read** method will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **ReadLine** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.ReadLine Overload List](#)

InternetServer.ReadLine Method (String, Int32)

Read up to a line of data from the client and return it in a string buffer.

[Visual Basic]

```
Overloads Public Function ReadLine( _  
    ByRef buffer As String, _  
    ByVal length As Integer _  
) As Boolean
```

[C#]

```
public bool ReadLine(  
    ref string buffer,  
    int length  
);
```

Parameters

buffer

A string which will contain the data read from the client.

length

An integer value which specifies the maximum number of bytes of data to read.

Return Value

This method returns a boolean value which specifies if a line of data has been read. A value of **true** indicates a line of data has been read. If an error occurs or there is no more data available to read, then the method will return **false**. It is possible for data to be returned in the string buffer even if the return value is **false**.

Applications should check the length of the string after the method returns to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Remarks

The **ReadLine** method reads data from the client up to the specified number of bytes or until an end-of-line character sequence is encountered. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a single line of text data in a string variable. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point; the string will not contain the carriage-return or linefeed characters.

There are some limitations when using the **ReadLine** method. The method should only be used to read text, never binary data. In particular, it will discard nulls, linefeed and carriage return control characters. This method will force the calling thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection.

The **Read** and **ReadLine** methods can be intermixed, however be aware that the **Read** method will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **ReadLine** method outside of an event handler, you must explicitly specify the client handle.

See Also

InternetServer.Reject Method

Rejects a connection request from a client.

Overload List

Rejects a connection request from a client.

```
public bool Reject();
```

Rejects a connection request from a client.

```
public bool Reject(int);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Reject Method ()

Rejects a connection request from a client.

[Visual Basic]

Overloads Public Function Reject() As Boolean

[C#]

public bool Reject();

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The **Reject** method rejects a pending client connection and the remote host will see this as the connection being aborted. If there are no pending client connections at the time, this method will immediately return with an error indicating that the operation would cause the thread to block.

This method is typically called from within the **OnAccept** event handler when the application determines that it does not wish to accept the incoming client connection.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Reject** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Reject Overload List](#)

InternetServer.Reject Method (Int32)

Rejects a connection request from a client.

[Visual Basic]

```
Overloads Public Function Reject( _  
    ByVal handle As Integer _  
) As Boolean
```

[C#]

```
public bool Reject(  
    int handle  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The **Reject** method rejects a pending client connection and the remote host will see this as the connection being aborted. If there are no pending client connections at the time, this method will immediately return with an error indicating that the operation would cause the thread to block.

This method is typically called from within the **OnAccept** event handler when the application determines that it does not wish to accept the incoming client connection.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Reject Overload List](#)

InternetServer.Reset Method

Reset the internal state of the object, resetting all properties to their default values.

[Visual Basic]

```
Public Sub Reset()
```

[C#]

```
public void Reset();
```

Remarks

The **Reset** method returns the object to its default state. If a server has been started, it will be stopped and any active client connections will be terminated. All properties will be reset to their default values.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Resolve Method

Resolves a host name to a host IP address.

[Visual Basic]

```
Public Function Resolve( _  
    ByVal hostName As String, _  
    ByRef hostAddress As String _  
) As Boolean
```

[C#]

```
public bool Resolve(  
    string hostName,  
    ref string hostAddress  
);
```

Parameters

hostName

A string which specifies the host name to be resolved.

hostAddress

A string which will contain the Internet address for the specified host.

Return Value

This method returns a boolean value. If the host name can be resolved, the return value is **true**. If the host name cannot be resolved, the return value is **false**. To get extended error information, check the value of the **LastError** property.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Restart Method

Restarts the server and terminates all active client connections.

[Visual Basic]

```
Public Function Restart() As Boolean
```

[C#]

```
public bool Restart();
```

Return Value

A boolean value which specifies if the server was restarted. A return value of **true** specifies that the server has been successfully restarted. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Restart** method terminates all active client connections, recreates a new listening socket bound to the same address and port number, and then resumes accepting new client connections.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Resume Method

Resume accepting new client connections.

[Visual Basic]

```
Public Function Resume() As Boolean
```

[C#]

```
public bool Resume();
```

Return Value

A boolean value which specifies if the server has resumed accepting client connections. A return value of **true** specifies that the operation was successful. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Resume** method instructs the server to resume accepting new client connections. Any pending client connections that were requested while the server was suspended will be accepted.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Start Method

Start listening for client connections.

Overload List

Start listening for client connections.

```
public bool Start();
```

Start listening for client connections on the specified port number.

```
public bool Start(int);
```

Start listening for client connections on the specified IP address and port number.

```
public bool Start(string,int);
```

Start listening for client connections on the specified IP address and port number.

```
public bool Start(string,int,int);
```

Start listening for client connections on the specified IP address and port number.

```
public bool Start(string,int,int,int);
```

Start listening for client connections on the specified IP address and port number.

```
public bool Start(string,int,int,int,int);
```

Start listening for client connections on the specified IP address and port number.

```
public bool Start(string,int,int,int,int,ServerOptions);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [Backlog Property](#) | [MaxClients Property](#) | [Options Property](#) | [ServerAddress Property](#) | [ServerPort Property](#) | [Timeout Property](#)

InternetServer.Start Method ()

Start listening for client connections.

[Visual Basic]

Overloads Public Function Start() As Boolean

[C#]

public bool Start();

Return Value

A boolean value which specifies if the server has been started. A return value of **true** specifies that the operation was successful. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Start** method begins listening for client connections on the default local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread.

The value of the **Backlog** property will determine the default size of the queue for incoming client connections. The value of the **MaxClients** property will determine the maximum number of clients that may connect to the server. The value of the **Options** property will determine the default options used when starting the server. The value of the **ServerAddress** and **ServerPort** properties will determine the address and port number that the server will accept client connections on. The value of the **Timeout** property will determine the default timeout period.

Example

The following example demonstrates creating an instance of the **InternetServer** class object and starting a server using the **Start** method.

```
Dim Server As SocketTools.InternetServer

Server = New SocketTools.InternetServer
Server.ServerAddress = TextBox1.Text.Trim()
Server.ServerPort = Val(TextBox2.Text)

If Server.Start() Then
    StatusBar1.Text = "The server has started listening for connections"
Else
    StatusBar1.Text = "The server could not be started"
End If
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Start Overload List](#) | [Backlog Property](#) | [MaxClients Property](#) | [Options Property](#) | [ServerAddress Property](#) | [ServerPort Property](#) | [Timeout Property](#)

InternetServer.Start Method (Int32)

Start listening for client connections on the specified port number.

[Visual Basic]

```
Overloads Public Function Start( _  
    ByVal LocalPort As Integer _  
) As Boolean
```

[C#]

```
public bool Start(  
    int LocalPort  
);
```

Parameters

localPort

An integer value which specifies the port number that the server should use when listening for incoming client connections. Valid port numbers are in the range of 1 through 65535.

Return Value

A boolean value which specifies if the server has been started. A return value of **true** specifies that the operation was successful. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Start** method begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread.

The value of the **Backlog** property will determine the default size of the queue for incoming client connections. The value of the **MaxClients** property will determine the maximum number of clients that may connect to the server. The value of the **Options** property will determine the default options used when starting the server. The value of the **ServerAddress** property will determine the address that the server will accept client connections on. The value of the **Timeout** property will determine the default timeout period.

Example

The following example demonstrates creating an instance of the **InternetServer** class object and starting a server using the [Start](#) method.

```
Dim Server As SocketTools.InternetServer  
Dim nLocalPort As Integer  
  
nLocalPort = Val(TextBox1.Text)  
  
If Server.Start(nLocalPort) Then  
    StatusBar1.Text = "The server has started listening for connections"  
Else  
    StatusBar1.Text = "The server could not be started"  
End If
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Start Overload List](#) | [Backlog Property](#) | [MaxClients Property](#) | [Options Property](#) | [ServerAddress Property](#) | [Timeout Property](#)

InternetServer.Start Method (String, Int32)

Start listening for client connections on the specified IP address and port number.

[Visual Basic]

```
Overloads Public Function Start( _  
    ByVal LocalAddress As String, _  
    ByVal LocalPort As Integer _  
) As Boolean
```

[C#]

```
public bool Start(  
    string LocalAddress,  
    int LocalPort  
);
```

Parameters

localAddress

A string value which specifies the IP address of the network adapter that the control should use when listening for connection requests. If this is an empty string or the special address "0.0.0.0" is specified, the server will listen for connection on all valid network interfaces configured for the local system.

localPort

An integer value which specifies the port number that the server should use when listening for incoming client connections. Valid port numbers are in the range of 1 through 65535.

Return Value

A boolean value which specifies if the server has been started. A return value of **true** specifies that the operation was successful. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Start** method begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread.

The value of the **Backlog** property will determine the default size of the queue for incoming client connections. The value of the **MaxClients** property will determine the maximum number of clients that may connect to the server. The value of the **Options** property will determine the default options used when starting the server. The value of the **Timeout** property will determine the default timeout period.

Example

The following example demonstrates creating an instance of the **InternetServer** class object and starting a server using the [Start](#) method.

```
Dim Server As SocketTools.InternetServer  
Dim strLocalAddress As String  
Dim nLocalPort As Integer
```

```
Server = New SocketTools.InternetServer
```

```
strLocalAddress = TextBox1.Text.Trim()  
nLocalPort = Val(TextBox2.Text)
```

```
If Server.Start(strLocalAddress, nLocalPort) Then  
    StatusBar1.Text = "The server has started listening for connections"
```

```
Else  
    StatusBar1.Text = "The server could not be started"  
End If
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Start Overload List](#) | [Backlog Property](#) | [MaxClients Property](#) | [Options Property](#) | [Timeout Property](#)

InternetServer.Start Method (String, Int32, Int32)

Start listening for client connections on the specified IP address and port number.

[Visual Basic]

```
Overloads Public Function Start( _  
    ByVal LocalAddress As String, _  
    ByVal LocalPort As Integer, _  
    ByVal maxClients As Integer _  
) As Boolean
```

[C#]

```
public bool Start(  
    string LocalAddress,  
    int LocalPort,  
    int maxClients  
);
```

Parameters

localAddress

A string value which specifies the IP address of the network adapter that the control should use when listening for connection requests. If this is an empty string or the special address "0.0.0.0" is specified, the server will listen for connection on all valid network interfaces configured for the local system.

localPort

An integer value which specifies the port number that the server should use when listening for incoming client connections. Valid port numbers are in the range of 1 through 65535.

maxClients

An integer value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of active client connections that may be established with the server. This value can be adjusted after the server has been created by calling the **Throttle** method.

Return Value

A boolean value which specifies if the server has been started. A return value of **true** specifies that the operation was successful. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Start** method begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread.

The value of the **Backlog** property will determine the default size of the queue for incoming client connections. The value of the **Options** property will determine the default options used when starting the server. The value of the **Timeout** property will determine the default timeout period.

Example

The following example demonstrates creating an instance of the **InternetServer** class object and starting a server using the [Start](#) method.

```
Dim Server As SocketTools.InternetServer  
Dim strLocalAddress As String  
Dim nLocalPort As Integer
```

```
Dim nMaxClients As Integer
```

```
Server = New SocketTools.InternetServer
```

```
strLocalAddress = TextBox1.Text.Trim()
```

```
nLocalPort = Val(TextBox2.Text)
```

```
nMaxClients = Val(TextBox4.Text)
```

```
If Server.Start(strLocalAddress, nLocalPort, nMaxClients) Then
```

```
    StatusBar1.Text = "The server has started listening for connections"
```

```
Else
```

```
    StatusBar1.Text = "The server could not be started"
```

```
End If
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Start Overload List](#) | [Backlog Property](#) | [Options Property](#) | [Timeout Property](#)

InternetServer.Start Method (String, Int32, Int32, Int32)

Start listening for client connections on the specified IP address and port number.

[Visual Basic]

```
Overloads Public Function Start( _  
    ByVal LocalAddress As String, _  
    ByVal LocalPort As Integer, _  
    ByVal backlog As Integer, _  
    ByVal maxClients As Integer _  
) As Boolean
```

[C#]

```
public bool Start(  
    string LocalAddress,  
    int LocalPort,  
    int backlog,  
    int maxClients  
);
```

Parameters

localAddress

A string value which specifies the IP address of the network adapter that the control should use when listening for connection requests. If this is an empty string or the special address "0.0.0.0" is specified, the server will listen for connection on all valid network interfaces configured for the local system.

localPort

An integer value which specifies the port number that the server should use when listening for incoming client connections. Valid port numbers are in the range of 1 through 65535.

backlog

An integer value which specifies the maximum size of the queue used to manage pending connections to the service. If the argument is set to value which exceeds the maximum size for the underlying service provider, it will be silently adjusted to the nearest legal value. On Windows workstations, the maximum backlog value is 5. On Windows servers, the maximum value is 200.

maxClients

An integer value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of active client connections that may be established with the server. This value can be adjusted after the server has been created by calling the **Throttle** method.

Return Value

A boolean value which specifies if the server has been started. A return value of **true** specifies that the operation was successful. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Start** method begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread.

The value of the **Options** property will determine the default options used when starting the server. The value of the **Timeout** property will determine the default timeout period.

Example

The following example demonstrates creating an instance of the **InternetServer** class object and starting a server using the [Start](#) method.

```
Dim Server As SocketTools.InternetServer
Dim strLocalAddress As String
Dim nLocalPort As Integer
Dim nBacklog As Integer
Dim nMaxClients As Integer

Server = New SocketTools.InternetServer

strLocalAddress = TextBox1.Text.Trim()
nLocalPort = Val(TextBox2.Text)
nBacklog = Val(TextBox3.Text)
nMaxClients = Val(TextBox4.Text)

If Server.Start(strLocalAddress, nLocalPort, nBacklog, nMaxClients) Then
    StatusBar1.Text = "The server has started listening for connections"
Else
    StatusBar1.Text = "The server could not be started"
End If
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Start Overload List](#) | [Options Property](#) | [Timeout Property](#)

InternetServer.Start Method (String, Int32, Int32, Int32, Int32)

Start listening for client connections on the specified IP address and port number.

[Visual Basic]

```
Overloads Public Function Start( _  
    ByVal LocalAddress As String, _  
    ByVal LocalPort As Integer, _  
    ByVal backlog As Integer, _  
    ByVal maxClients As Integer, _  
    ByVal timeout As Integer _  
) As Boolean
```

[C#]

```
public bool Start(  
    string LocalAddress,  
    int LocalPort,  
    int backlog,  
    int maxClients,  
    int timeout  
);
```

Parameters

localAddress

A string value which specifies the IP address of the network adapter that the control should use when listening for connection requests. If this is an empty string or the special address "0.0.0.0" is specified, the server will listen for connection on all valid network interfaces configured for the local system.

localPort

An integer value which specifies the port number that the server should use when listening for incoming client connections. Valid port numbers are in the range of 1 through 65535.

backlog

An integer value which specifies the maximum size of the queue used to manage pending connections to the service. If the argument is set to value which exceeds the maximum size for the underlying service provider, it will be silently adjusted to the nearest legal value. On Windows workstations, the maximum backlog value is 5. On Windows servers, the maximum value is 200.

maxClients

An integer value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of active client connections that may be established with the server. This value can be adjusted after the server has been created by calling the **Throttle** method.

timeout

An integer value which specifies the number of seconds the control will wait for a network operation to complete. The default timeout period of 20 seconds is sufficient for most applications.

Return Value

A boolean value which specifies if the server has been started. A return value of **true** specifies that the operation was successful. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Start** method begins listening for client connections on the specified local address and port number. The

server is started in its own thread and manages the client sessions independently of the calling thread.

The value of the **Options** property determines the default options that will be used when starting the server.

Example

The following example demonstrates creating an instance of the **InternetServer** class object and starting a server using the [Start](#) method.

```
Dim Server As SocketTools.InternetServer
Dim strLocalAddress As String
Dim nLocalPort As Integer
Dim nBacklog As Integer
Dim nMaxClients As Integer
Dim nTimeout As Integer

Server = New SocketTools.InternetServer

strLocalAddress = TextBox1.Text.Trim()
nLocalPort = Val(TextBox2.Text)
nBacklog = Val(TextBox3.Text)
nMaxClients = Val(TextBox4.Text)
nTimeout = Val(TextBox5.Text)

If Server.Start(strLocalAddress, nLocalPort, nBacklog, nMaxClients, nTimeout) Then
    StatusBar1.Text = "The server has started listening for connections"
Else
    StatusBar1.Text = "The server could not be started"
End If
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Start Overload List](#) | [Options Property](#)

InternetServer.Start Method (String, Int32, Int32, Int32, Int32, ServerOptions)

Start listening for client connections on the specified IP address and port number.

[Visual Basic]

```
Overloads Public Function Start( _  
    ByVal LocalAddress As String, _  
    ByVal LocalPort As Integer, _  
    ByVal backlog As Integer, _  
    ByVal maxClients As Integer, _  
    ByVal timeout As Integer, _  
    ByVal options As ServerOptions _  
) As Boolean
```

[C#]

```
public bool Start(  
    string LocalAddress,  
    int LocalPort,  
    int backlog,  
    int maxClients,  
    int timeout,  
    ServerOptions options  
);
```

Parameters

localAddress

A string value which specifies the IP address of the network adapter that the control should use when listening for connection requests. If this is an empty string or the special address "0.0.0.0" is specified, the server will listen for connection on all valid network interfaces configured for the local system.

localPort

An integer value which specifies the port number that the server should use when listening for incoming client connections. Valid port numbers are in the range of 1 through 65535.

backlog

An integer value which specifies the maximum size of the queue used to manage pending connections to the service. If the argument is set to value which exceeds the maximum size for the underlying service provider, it will be silently adjusted to the nearest legal value. On Windows workstations, the maximum backlog value is 5. On Windows servers, the maximum value is 200.

maxClients

An integer value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of active client connections that may be established with the server. This value can be adjusted after the server has been created by calling the **Throttle** method.

timeout

An integer value which specifies the number of seconds the control will wait for a network operation to complete. The default timeout period of 20 seconds is sufficient for most applications.

options

One or more of the [ServerOptions](#) enumeration flags.

Return Value

A boolean value which specifies if the server has been started. A return value of **true** specifies that the

operation was successful. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Start** method begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread.

Example

The following example demonstrates creating an instance of the **InternetServer** class object and starting a server using the [Start](#) method.

```
Dim Server As SocketTools.InternetServer
Dim strLocalAddress As String
Dim nLocalPort As Integer
Dim nBacklog As Integer
Dim nMaxClients As Integer
Dim nTimeout As Integer

Server = New SocketTools.InternetServer

strLocalAddress = TextBox1.Text.Trim()
nLocalPort = Val(TextBox2.Text)
nBacklog = Val(TextBox3.Text)
nMaxClients = Val(TextBox4.Text)
nTimeout = Val(TextBox5.Text)

If Server.Start(strLocalAddress, nLocalPort, nBacklog, nMaxClients, nTimeout) Then
    StatusBar1.Text = "The server has started listening for connections"
Else
    StatusBar1.Text = "The server could not be started"
End If
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Start Overload List](#)

InternetServer.Stop Method

Stop listening for new client connections and terminate all active clients already connected to the server.

[Visual Basic]

```
Public Function Stop() As Boolean
```

[C#]

```
public bool Stop();
```

Return Value

A boolean value which specifies if the server was stopped. A return value of **true** specifies that the server has been successfully stopped. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Stop** method instructs the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session.

If this method is called when there is one or more clients connected to the server, it will signal each client thread to terminate and then wait for the server thread to terminate. As the client sessions are terminated, the **OnDisconnect** event handler will not be invoked. If you wish to ensure that all clients are disconnected normally before stopping the server, call the **Suspend** method with the **Disconnect** option and then stop the server after the last client has disconnected.

After the server has been terminated, the closed listening socket will go into a TIME-WAIT state which prevents an application from reusing the same address and port number bound to that socket for a brief period of time, typically two to four minutes. This is normal behavior designed to prevent delayed or misrouted packets of data from being read by a subsequent connection. To immediately start a new server using the same local address and port number, set the **ReuseAddress** property to a value of **true**.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [Restart Method](#) | [Resume Method](#) | [Start Method](#) | [Throttle Method](#)

InternetServer.Suspend Method

Suspend accepting new client connections.

Overload List

Suspend accepting new client connections.

```
public bool Suspend();
```

Suspend accepting new client connections with additional options.

```
public bool Suspend(SuspendOptions);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Suspend Method ()

Suspend accepting new client connections.

[Visual Basic]

Overloads Public Function Suspend() As Boolean

[C#]

public bool Suspend();

Return Value

A boolean value which specifies if the server was suspended. A return value of **true** specifies that the server has suspended accepting new client connections. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Suspend** method instructs the server to suspend accepting new client connections. By default, any incoming client connections will be queued up to the maximum backlog value specified when the server was started. To resume accepting client connections, call the **Resume** method.

It is not recommended that you leave a server in a suspended state for extended periods of time. Once the connection backlog queue has filled, subsequent incoming client connections will be rejected. If you wish to suspend the server for more than a few seconds, call the overloaded version of this method and specify the **Reject** option. This will reject all incoming client connections to the server, rather than forcing clients to wait.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Suspend Overload List](#)

InternetServer.Suspend Method (SuspendOptions)

Suspend accepting new client connections with additional options.

[Visual Basic]

```
Overloads Public Function Suspend( _  
    ByVal options As SuspendOptions _  
) As Boolean
```

[C#]

```
public bool Suspend(  
    SuspendOptions options  
);
```

Parameters

options

One or more of the [SuspendOptions](#) enumeration flags.

Return Value

A boolean value which specifies if the server was suspended. A return value of **true** specifies that the server has suspended accepting new client connections. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Suspend** method instructs the server to suspend accepting new client connections. By default, any incoming client connections will be queued up to the maximum backlog value specified when the server was started. To resume accepting client connections, call the **Resume** method.

If the **Disconnect** option is specified, the server will signal each client to disconnect and will stop accepting new connections. The **OnDisconnect** event handler will be invoked for each client that disconnects from the server. If the **Wait** option is also specified, this method will wait until the last client has disconnected from the server before returning to the caller. If there are a large number of clients connected to the server, this process may cause the application to block for an extended period of time and appear to be non-responsive to the user. For this reason, you should not specify the **Wait** option if the method is being called from the application's main UI thread.

To perform a graceful shutdown of the server, it is recommended that you call the **Suspend** method with the **Reject** and **Disconnect** options. This will allow each client to disconnect from the server and the server will reject any new incoming connections. After the last client has disconnected from the server, the **OnIdle** event handler will be invoked and the application can call the **Stop** method to complete the shutdown process.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Suspend Overload List](#) | [SuspendOptions Enumeration](#)

InternetServer.Throttle Method

Limit the maximum number of client connections.

Overload List

Limit the maximum number of client connections.

```
public bool Throttle(int);
```

Limit the maximum number of client connections and connections per IP address.

```
public bool Throttle(int,int);
```

Limit the maximum number of client connections, connections per IP address and connection rate.

```
public bool Throttle(int,int,int);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Throttle Method (Int32)

Limit the maximum number of client connections.

[Visual Basic]

```
Overloads Public Function Throttle( _  
    ByVal maxClients As Integer _  
) As Boolean
```

[C#]

```
public bool Throttle(  
    int maxClients  
);
```

Parameters

maxClients

An integer value that specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections.

Return Value

A boolean value which specifies if the method was successful. A return value of **true** indicates success. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **Start** method is called with the maximum number of clients set to 100, and then the **Throttle** method is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Throttle Overload List](#)

InternetServer.Throttle Method (Int32, Int32)

Limit the maximum number of client connections and connections per IP address.

[Visual Basic]

```
Overloads Public Function Throttle( _  
    ByVal maxClients As Integer, _  
    ByVal maxClientsPerAddress As Integer _  
) As Boolean
```

[C#]

```
public bool Throttle(  
    int maxClients,  
    int maxClientsPerAddress  
);
```

Parameters

maxClients

An integer value that specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections.

maxClientsPerAddress

An integer value that specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is no limit on the number of client connections per address.

Return Value

A boolean value which specifies if the method was successful. A return value of **true** indicates success. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **Start** method is called with the maximum number of clients set to 100, and then the **Throttle** method is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Throttle Overload List](#)

InternetServer.Throttle Method (Int32, Int32, Int32)

Limit the maximum number of client connections, connections per IP address and connection rate.

[Visual Basic]

```
Overloads Public Function Throttle( _  
    ByVal maxClients As Integer, _  
    ByVal maxClientsPerAddress As Integer, _  
    ByVal connectionRate As Integer _  
) As Boolean
```

[C#]

```
public bool Throttle(  
    int maxClients,  
    int maxClientsPerAddress,  
    int connectionRate  
);
```

Parameters

maxClients

An integer value that specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections.

maxClientsPerAddress

An integer value that specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is no limit on the number of client connections per address.

connectionRate

An integer value that specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate.

Return Value

A boolean value which specifies if the method was successful. A return value of **true** indicates success. If an error occurs, the method returns **false** and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **Start** method is called with the maximum number of clients set to 100, and then the **Throttle** method is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will accept incoming client connection requests. For example, setting this parameter to a value of 1000 would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being

rejected.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Throttle Overload List](#)

InternetServer.Uninitialize Method

Uninitialize the class library and release any resources allocated for the server.

[Visual Basic]

```
Public Sub Uninitialize()
```

[C#]

```
public void Uninitialize();
```

Remarks

The **Uninitialize** method terminates any active connection, releases resources allocated for the server and unloads the networking library. After this method has been called, no further network operations may be performed until the class instance has been re-initialized.

If the **Initialize** method is explicitly called by the application, it should be matched by a call to the **Uninitialize** method when that instance of the class is no longer needed.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [Initialize Method](#)

InternetServer.Unlock Method

Unlock the server and allow other server threads to resume execution.

[Visual Basic]

```
Public Function Unlock() As Boolean
```

[C#]

```
public bool Unlock();
```

Return Value

A boolean value which specifies if the server was unlocked. A return value of **true** specifies that the server was unlocked, and the threads being managed by the server have resumed normal execution. A return value of **false** indicates that the server could not be unlocked, typically because a potential deadlock was detected.

Remarks

The **Unlock** method releases the lock on the server and allows any blocked threads to resume execution. Only one server may be locked at any one time, and only the thread which established the lock can unlock the server.

Every time the **Lock** method is called, an internal lock counter is incremented, and the lock will not be released until the lock count drops to zero. This means that each call to the **Lock** method must be matched by an equal number of calls to the **Unlock** method. Failure to do so will result in the server becoming non-responsive as it remains in a locked state.

The program should always check the return value from this method, and should never assume that the lock has been released. If a potential deadlock situation is detected, this method will fail and return a value of false.

The **IsLocked** property can be used to determine if the server has been locked.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [Lock Method](#) | [IsLocked Property](#)

InternetServer.Write Method

Write one or more bytes of data to a client.

Overload List

Write one or more bytes of data to a client.

```
public int Write(byte[]);
```

Write one or more bytes of data to a client.

```
public int Write(byte[],int);
```

Write one or more bytes of data to a client.

```
public int Write(int,byte[]);
```

Write one or more bytes of data to a client.

```
public int Write(int,byte[],int);
```

Write a string of characters to a client.

```
public int Write(int,string);
```

Write a string of characters to a client.

```
public int Write(int,string,int);
```

Write a string of characters to a client.

```
public int Write(string);
```

Write a string of characters to a client.

```
public int Write(string,int);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.Write Method (Byte[])

Write one or more bytes of data to a client.

[Visual Basic]

```
Overloads Public Function Write( _  
    ByVal buffer As Byte() _  
) As Integer
```

[C#]

```
public int Write(  
    byte[] buffer  
);
```

Parameters

buffer

A byte array that contains the data to be written to the client.

Return Value

An integer value which specifies the number of bytes actually written to the client. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Write** method sends one or more bytes of data to a client. If there is enough room in the client socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space the method will block the current thread until the data can be sent.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Write** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Write Overload List](#)

InternetServer.Write Method (Byte[], Int32)

Write one or more bytes of data to a client.

[Visual Basic]

```
Overloads Public Function Write( _  
    ByVal buffer As Byte(), _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Write(  
    byte[] buffer,  
    int length  
);
```

Parameters

buffer

A byte array that contains the data to be written to the client.

length

An integer value which specifies the maximum number of bytes of data to write. This value cannot be larger than the size of the buffer specified by the caller.

Return Value

An integer value which specifies the number of bytes actually written to the client. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Write** method sends one or more bytes of data to a client. If there is enough room in the client socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space the method will block the current thread until the data can be sent.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Write** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Write Overload List](#)

InternetServer.Write Method (Int32, Byte[])

Write one or more bytes of data to a client.

[Visual Basic]

```
Overloads Public Function Write( _  
    ByVal handle As Integer, _  
    ByVal buffer As Byte() _  
) As Integer
```

[C#]

```
public int Write(  
    int handle,  
    byte[] buffer  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A byte array that contains the data to be written to the client.

Return Value

An integer value which specifies the number of bytes actually written to the client. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Write** method sends one or more bytes of data to the specified client. If there is enough room in the client socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space the method will block the current thread until the data can be sent.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Write Overload List](#)

InternetServer.Write Method (Int32, Byte[], Int32)

Write one or more bytes of data to a client.

[Visual Basic]

```
Overloads Public Function Write( _  
    ByVal handle As Integer, _  
    ByVal buffer As Byte(), _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Write(  
    int handle,  
    byte[] buffer,  
    int length  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A byte array that contains the data to be written to the client.

length

An integer value which specifies the maximum number of bytes of data to write. This value cannot be larger than the size of the buffer specified by the caller.

Return Value

An integer value which specifies the number of bytes actually written to the client. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Write** method sends one or more bytes of data to the specified client. If there is enough room in the client socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space the method will block the current thread until the data can be sent.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Write Overload List](#) | [Broadcast Method](#)

InternetServer.Write Method (Int32, String)

Write a string of characters to a client.

[Visual Basic]

```
Overloads Public Function Write( _  
    ByVal handle As Integer, _  
    ByVal buffer As String _  
) As Integer
```

[C#]

```
public int Write(  
    int handle,  
    string buffer  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A string which contains the data to be written to the client.

Return Value

An integer value which specifies the number of characters actually written to the client. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Write** method sends a string of characters to a client. If there is enough room in the socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space the method will block the current thread until the data can be sent.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Write Overload List](#)

InternetServer.Write Method (Int32, String, Int32)

Write a string of characters to a client.

[Visual Basic]

```
Overloads Public Function Write( _  
    ByVal handle As Integer, _  
    ByVal buffer As String, _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Write(  
    int handle,  
    string buffer,  
    int length  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A string which contains the data to be written to the client.

length

An integer value which specifies the maximum number of characters to write. This value cannot be larger than the length of the string specified by the caller.

Return Value

An integer value which specifies the number of characters actually written to the client. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Write** method sends a string of characters to a client. If there is enough room in the socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space the method will block the current thread until the data can be sent.

The string will be converted to an array of bytes before being written to the socket. By default, the character encoding used will be for the current locale. Depending on the contents of the string, the number of bytes written may be different than the string length specified. This is because the conversion from Unicode to a byte array may result in a multi-byte character sequence.

You should never use strings to read and write binary data. Always use byte arrays to ensure that no character conversion is performed.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Write Overload List](#)

InternetServer.Write Method (String)

Write a string of characters to a client.

[Visual Basic]

```
Overloads Public Function Write( _  
    ByVal buffer As String _  
) As Integer
```

[C#]

```
public int Write(  
    string buffer  
);
```

Parameters

buffer

A string which contains the data to be written to the client.

Return Value

An integer value which specifies the number of characters actually written to the client. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Write** method sends a string of characters to a client. If there is enough room in the socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space the method will block the current thread until the data can be sent.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Write** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Write Overload List](#)

InternetServer.Write Method (String, Int32)

Write a string of characters to a client.

[Visual Basic]

```
Overloads Public Function Write( _  
    ByVal buffer As String, _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Write(  
    string buffer,  
    int length  
);
```

Parameters

buffer

A string which contains the data to be written to the client.

length

An integer value which specifies the maximum number of characters to write. This value cannot be larger than the length of the string specified by the caller.

Return Value

An integer value which specifies the number of characters actually written to the client. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Write** method sends a string of characters to a client. If there is enough room in the socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space the method will block the current thread until the data can be sent.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **Write** method outside of an event handler, you must explicitly specify the client handle.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.Write Overload List](#)

InternetServer.WriteLine Method

Send an empty line of text to a client, terminated by a carriage-return and linefeed.

Overload List

Send an empty line of text to a client, terminated by a carriage-return and linefeed.

```
public bool WriteLine();
```

Send an empty line of text to a client, terminated by a carriage-return and linefeed.

```
public bool WriteLine(int);
```

Send a line of text to a client, terminated by a carriage-return and linefeed.

```
public bool WriteLine(int,string);
```

Send a line of text to a client, terminated by a carriage-return and linefeed.

```
public bool WriteLine(int,string,ref int);
```

Send a line of text to a client, terminated by a carriage-return and linefeed.

```
public bool WriteLine(string);
```

Send a line of text to a client, terminated by a carriage-return and linefeed.

```
public bool WriteLine(string,ref int);
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.WriteLine Method ()

Send an empty line of text to a client, terminated by a carriage-return and linefeed.

[Visual Basic]

Overloads Public Function WriteLine() As Boolean

[C#]

public bool WriteLine();

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

The **WriteLine** method will send an empty line of text to the client, terminated by a carriage-return and linefeed. Calling this method will force the calling thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **WriteLine** method outside of an event handler, you must explicitly specify the client handle.

The **Write** and **WriteLine** methods can be safely intermixed.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.WriteLine Overload List](#)

InternetServer.WriteLine Method (Int32)

Send an empty line of text to a client, terminated by a carriage-return and linefeed.

[Visual Basic]

```
Overloads Public Function WriteLine( _  
    ByVal handle As Integer _  
) As Boolean
```

[C#]

```
public bool WriteLine(  
    int handle  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

The **WriteLine** method will send an empty line of text to the specified client, terminated by a carriage-return and linefeed. Calling this method will force the calling thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection.

The **Write** and **WriteLine** methods can be safely intermixed.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.WriteLine Overload List](#)

InternetServer.WriteLine Method (Int32, String)

Send a line of text to a client, terminated by a carriage-return and linefeed.

[Visual Basic]

```
Overloads Public Function WriteLine( _  
    ByVal handle As Integer, _  
    ByVal buffer As String _  
) As Boolean
```

[C#]

```
public bool WriteLine(  
    int handle,  
    string buffer  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A string which contains the data that will be sent to the specified client. The data will always be terminated with a carriage-return and linefeed control character sequence. If the string is empty, then a only a carriage-return and linefeed are written to the socket. Note that if the string contains a null character, any data that follows the null character will be discarded.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

The **WriteLine** method should only be used to send text, never binary data. In particular, this method will discard any data that follows a null character and will append linefeed and carriage return control characters to the data stream. Calling this method will force the current thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection.

The **Write** and **WriteLine** methods can be safely intermixed.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.WriteLine Overload List](#)

InternetServer.WriteLine Method (Int32, String, Int32)

Send a line of text to a client, terminated by a carriage-return and linefeed.

[Visual Basic]

```
Overloads Public Function WriteLine( _  
    ByVal handle As Integer, _  
    ByVal buffer As String, _  
    ByRef length As Integer _  
) As Boolean
```

[C#]

```
public bool WriteLine(  
    int handle,  
    string buffer,  
    ref int length  
);
```

Parameters

handle

An integer value which specifies the handle to the client session.

buffer

A string which contains the data that will be sent to the specified client. The data will always be terminated with a carriage-return and linefeed control character sequence. If the string is empty, then a only a carriage-return and linefeed are written to the client. Note that if the string contains a null character, any data that follows the null character will be discarded.

length

An integer value which specifies the maximum number of characters to write. This value cannot be larger than the length of the string specified by the caller.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

The **WriteLine** method should only be used to send text, never binary data. In particular, this method will discard any data that follows a null character and will append linefeed and carriage return control characters to the data stream. Calling this method will force the current thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection.

The string will be converted to an array of bytes before being written to the socket. By default, the character encoding used will be for the current locale. Depending on the contents of the string, the number of bytes written may be different than the string length specified. This is because the conversion from Unicode to a byte array may result in a multi-byte character sequence.

You should never use strings to read and write binary data. Always use byte arrays to ensure that no character conversion is performed.

The **Write** and **WriteLine** methods can be safely intermixed.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.WriteLine Overload List](#)

InternetServer.WriteLine Method (String)

Send a line of text to a client, terminated by a carriage-return and linefeed.

[Visual Basic]

```
Overloads Public Function WriteLine( _  
    ByVal buffer As String _  
) As Boolean
```

[C#]

```
public bool WriteLine(  
    string buffer  
);
```

Parameters

buffer

A string which contains the data that will be sent to the client. The data will always be terminated with a carriage-return and linefeed control character sequence. If the string is empty, then a only a carriage-return and linefeed are written to the socket. Note that if the string contains a null character, any data that follows the null byte will be discarded.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

The **WriteLine** method should only be used to send text, never binary data. In particular, this method will discard any data that follows a null character and will append linefeed and carriage return control characters to the data stream. Calling this method will force the current thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **WriteLine** method outside of an event handler, you must explicitly specify the client handle.

The **Write** and **WriteLine** methods can be safely intermixed.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.WriteLine Overload List](#)

InternetServer.WriteLine Method (String, Int32)

Send a line of text to a client, terminated by a carriage-return and linefeed.

[Visual Basic]

```
Overloads Public Function WriteLine( _  
    ByVal buffer As String, _  
    ByRef length As Integer _  
) As Boolean
```

[C#]

```
public bool WriteLine(  
    string buffer,  
    ref int length  
);
```

Parameters

buffer

A string which contains the data that will be sent to the client. The data will always be terminated with a carriage-return and linefeed control character sequence. If the string is empty, then a only a carriage-return and linefeed are written to the socket. Note that if the string contains a null character, any data that follows the null byte will be discarded.

length

An integer value which specifies the maximum number of characters to write. This value cannot be larger than the length of the string specified by the caller.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

The **WriteLine** method should only be used to send text, never binary data. In particular, this method will discard any data that follows a null character and will append linefeed and carriage return control characters to the data stream. Calling this method will force the current thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection.

This implementation of the method can only be used within a class event handler, or a method that has been invoked from within an event handler. If you need to call the **WriteLine** method outside of an event handler, you must explicitly specify the client handle.

The **Write** and **WriteLine** methods can be safely intermixed.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#) | [InternetServer.WriteLine Overload List](#)

InternetServer Events

The events of the **InternetServer** class are listed below. For a complete list of **InternetServer** class members, see the [InternetServer Members](#) topic.

Public Instance Events

OnAccept	Occurs when a client attempts to establish a connection with the server.
OnCancel	Occurs when a blocking socket operation is canceled.
OnConnect	Occurs when a connection is established with the remote host.
OnDisconnect	Occurs when the remote host disconnects from the local system.
OnError	Occurs when an socket operation fails.
OnIdle	Occurs when there are no clients connected to the server.
OnRead	Occurs when data is available to be read from the client.
OnStart	Occurs when the server starts accepting connections.
OnStop	Occurs when the server stops accepting connections.
OnTimeout	Occurs when a blocking operation fails to complete before the timeout period elapses.
OnWrite	Occurs when data can be written to the client.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.OnAccept Event

Occurs when a client attempts to establish a connection with the server.

[Visual Basic]

Public Event OnAccept As [OnAcceptEventHandler](#)

[C#]

public event [OnAcceptEventHandler](#) OnAccept;

Event Data

The event handler receives an argument of type [InternetServer.AcceptEventArgs](#) containing data related to this event. The following **InternetServer.AcceptEventArgs** properties provide information specific to this event.

Property	Description
ClientAddress	Gets a value that specifies the Internet address of the current client session.
ClientPort	Gets a value that specifies the port number used by the current client session.
Handle	Gets a value that specifies the socket handle for the listening server.

Remarks

The **OnAccept** event occurs when a client attempts to connect to the local system. A connection is not actually established until it has been accepted by the server.

The **ClientAddress** or **ClientHost** properties may be used to determine the Internet address and host name of the remote host that is establishing the connection. To prevent the client from completing the connection, call the **Reject** method.

After the client connection has been established and the worker thread for that client session has started, the **OnConnect** event will fire.

User interface controls can only be accessed from the UI thread that created them, and attempting to update a control from another thread can result in the program becoming non-responsive or terminating abnormally. Because this event is generated in the context of the server thread, not the thread that created the class instance, you cannot directly modify a control from within this event handler. Instead, you must create a delegate and use the **Invoke** method to marshal invocations to the associated UI thread. For more information, refer to the documentation for the control.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.AcceptEventArgs Class

Provides data for the OnAccept event.

For a list of all members of this type, see [InternetServer.AcceptEventArgs Members](#).

System.Object

System.EventArgs

SocketTools.InternetServer.AcceptEventArgs

[Visual Basic]

Public Class InternetServer.AcceptEventArgs

Inherits EventArgs

[C#]

public class InternetServer.AcceptEventArgs : EventArgs

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

AcceptEventArgs specifies the socket handle for the server that should accept the incoming client connection.

The [OnAccept](#) event occurs when a remote host attempts to establish a connection with the local system.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[InternetServer.AcceptEventArgs Members](#) | [SocketTools Namespace](#) | [OnAccept Event \(SocketTools.InternetServer\)](#)

InternetServer.AcceptEventArgs Members

[InternetServer.AcceptEventArgs overview](#)

Public Instance Constructors

 InternetServer.AcceptEventArgs Constructor	Initializes a new instance of the InternetServer.AcceptEventArgs class.

Public Instance Properties

ClientAddress	Gets a value that specifies the Internet address of the current client session.
ClientPort	Gets a value that specifies the port number used by the current client session.
Handle	Gets a value that specifies the socket handle for the listening server.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetServer.AcceptEventArgs Class](#) | [SocketTools Namespace](#) | [OnAccept Event \(SocketTools.InternetServer\)](#)

InternetServer.AcceptEventArgs Constructor

Initializes a new instance of the [InternetServer.AcceptEventArgs](#) class.

[Visual Basic]

```
Public Sub New()
```

[C#]

```
public InternetServer.AcceptEventArgs();
```

See Also

[InternetServer.AcceptEventArgs Class](#) | [SocketTools Namespace](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InternetServer.AcceptEventArgs Properties

The properties of the **InternetServer.AcceptEventArgs** class are listed below. For a complete list of **InternetServer.AcceptEventArgs** class members, see the [InternetServer.AcceptEventArgs Members](#) topic.

Public Instance Properties

ClientAddress	Gets a value that specifies the Internet address of the current client session.
ClientPort	Gets a value that specifies the port number used by the current client session.
Handle	Gets a value that specifies the socket handle for the listening server.

See Also

[InternetServer.AcceptEventArgs Class](#) | [SocketTools Namespace](#) | [OnAccept Event \(SocketTools.InternetServer\)](#)

InternetServer.AcceptEventArgs.ClientAddress Property

Gets a value that specifies the Internet address of the current client session.

[Visual Basic]

```
Public ReadOnly Property ClientAddress As String
```

[C#]

```
public string ClientAddress {get;}
```

Remarks

The **ClientAddress** property will return the address of the client that is requesting the connection. The server application may use this information to determine if it wishes to accept or reject the client connection.

See Also

[InternetServer.AcceptEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.AcceptEventArgs.ClientPort Property

Gets a value that specifies the port number used by the current client session.

[Visual Basic]

```
Public ReadOnly Property ClientPort As Integer
```

[C#]

```
public int ClientPort {get;}
```

Remarks

The **ClientPort** property returns the port number that the client has used when establishing a connection with the server.

See Also

[InternetServer.AcceptEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.AcceptEventArgs.Handle Property

Gets a value that specifies the socket handle for the listening server.

[Visual Basic]

```
Public ReadOnly Property Handle As Integer
```

[C#]

```
public int Handle {get;}
```

Property Value

An integer value which specifies the server socket handle.

Remarks

The **Handle** property returns the socket handle for the server that generated the event. This value is used for identification purposes only and should not be used in conjunction with methods such as **Read** and **Write**, which may only be used with client handles.

See Also

[InternetServer.AcceptEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.OnCancel Event

Occurs when a blocking socket operation is canceled.

```
[Visual Basic]
Public Event OnCancel As OnCancelEventHandler
```

```
[C#]
public event OnCancelEventHandler OnCancel;
```

Event Data

The event handler receives an argument of type [InternetServer.CancelEventArgs](#) containing data related to this event. The following **InternetServer.CancelEventArgs** property provides information specific to this event.

Property	Description
Handle	Gets a value that specifies the socket handle for the client session.

Remarks

The **OnCancel** event is generated when a blocking socket operation, such as sending or receiving data, is canceled with the **Cancel** method.

User interface controls can only be accessed from the UI thread that created them, and attempting to update a control from another thread can result in the program becoming non-responsive or terminating abnormally. Because this event is generated in the context of the client thread, not the thread that created the class instance, you cannot directly modify a control from within this event handler. Instead, you must create a delegate and use the **Invoke** method to marshal invocations to the associated UI thread. For more information, refer to the documentation for the control.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.CancelEventArgs Class

Provides data for the OnCancel event.

For a list of all members of this type, see [InternetServer.CancelEventArgs Members](#).

System.Object

System.EventArgs

SocketTools.InternetServer.CancelEventArgs

[Visual Basic]

Public Class InternetServer.CancelEventArgs

Inherits EventArgs

[C#]

public class InternetServer.CancelEventArgs : EventArgs

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

CancelEventArgs specifies the socket handle for the current client session.

The [OnCancel](#) event occurs when a blocking network operation has been canceled.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[InternetServer.CancelEventArgs Members](#) | [SocketTools Namespace](#)

InternetServer.CancelEventArgs Members

[InternetServer.CancelEventArgs overview](#)

Public Instance Constructors

 InternetServer.CancelEventArgs Constructor	Initializes a new instance of the InternetServer.CancelEventArgs class.

Public Instance Properties

Handle	Gets a value that specifies the socket handle for the client session.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetServer.CancelEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.CancelEventArgs Constructor

Initializes a new instance of the [InternetServer.CancelEventArgs](#) class.

[Visual Basic]

```
Public Sub New()
```

[C#]

```
public InternetServer.CancelEventArgs();
```

See Also

[InternetServer.CancelEventArgs Class](#) | [SocketTools Namespace](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InternetServer.CancelEventArgs Properties

The properties of the **InternetServer.CancelEventArgs** class are listed below. For a complete list of **InternetServer.CancelEventArgs** class members, see the [InternetServer.CancelEventArgs Members](#) topic.

Public Instance Properties

Handle	Gets a value that specifies the socket handle for the client session.

See Also

[InternetServer.CancelEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.CancelEventArgs.Handle Property

Gets a value that specifies the socket handle for the client session.

[Visual Basic]

```
Public ReadOnly Property Handle As Integer
```

[C#]

```
public int Handle {get;}
```

Property Value

An integer value which specifies the client socket handle.

Remarks

The **Handle** property returns the socket handle for the client that generated the event. This handle can be used in conjunction with methods such as **Read** and **Write** to exchange data with the client.

See Also

[InternetServer.CancelEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.OnConnect Event

Occurs when a connection is established with the remote host.

[Visual Basic]

Public Event OnConnect As [OnConnectEventHandler](#)

[C#]

public event [OnConnectEventHandler](#) OnConnect;

Event Data

The event handler receives an argument of type [InternetServer.ConnectEventArgs](#) containing data related to this event. The following **InternetServer.ConnectEventArgs** property provides information specific to this event.

Property	Description
Handle	Gets a value that specifies the socket handle for the client session.

Remarks

The **OnConnect** event occurs when the client connection to the server has completed.

The **ClientAddress** property can be used to determine the IP address of the client which established the connection. To terminate the client connection, use the **Disconnect** method.

User interface controls can only be accessed from the UI thread that created them, and attempting to update a control from another thread can result in the program becoming non-responsive or terminating abnormally. Because this event is generated in the context of the client thread, not the thread that created the class instance, you cannot directly modify a control from within this event handler. Instead, you must create a delegate and use the **Invoke** method to marshal invocations to the associated UI thread. For more information, refer to the documentation for the control.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.ConnectEventArgs Class

Provides data for the OnConnect event.

For a list of all members of this type, see [InternetServer.ConnectEventArgs Members](#).

System.Object

System.EventArgs

SocketTools.InternetServer.ConnectEventArgs

[Visual Basic]

Public Class InternetServer.ConnectEventArgs

Inherits EventArgs

[C#]

public class InternetServer.ConnectEventArgs : EventArgs

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

ConnectEventArgs specifies the socket handle for the current client session.

The [OnConnect](#) event occurs when the client connection to the server has completed. The **Handle** property specifies the handle to the client socket that was allocated for the session. This handle can be used with methods such as **Read** and **Write** to exchange information with the client.

The **ClientAddress** property can be used to determine the IP address of the client which established the connection. To terminate the client connection, use the **Disconnect** method.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[InternetServer.ConnectEventArgs Members](#) | [SocketTools Namespace](#) | [OnConnect Event \(SocketTools.InternetServer\)](#)

InternetServer.ConnectEventArgs Members

[InternetServer.ConnectEventArgs overview](#)

Public Instance Constructors

 InternetServer.ConnectEventArgs Constructor	Initializes a new instance of the InternetServer.ConnectEventArgs class.

Public Instance Properties

Handle	Gets a value that specifies the socket handle for the client session.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetServer.ConnectEventArgs Class](#) | [SocketTools Namespace](#) | [OnConnect Event \(SocketTools.InternetServer\)](#)

InternetServer.ConnectEventArgs Constructor

Initializes a new instance of the [InternetServer.ConnectEventArgs](#) class.

[Visual Basic]

```
Public Sub New()
```

[C#]

```
public InternetServer.ConnectEventArgs();
```

See Also

[InternetServer.ConnectEventArgs Class](#) | [SocketTools Namespace](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InternetServer.ConnectEventArgs Properties

The properties of the **InternetServer.ConnectEventArgs** class are listed below. For a complete list of **InternetServer.ConnectEventArgs** class members, see the [InternetServer.ConnectEventArgs Members](#) topic.

Public Instance Properties

Handle	Gets a value that specifies the socket handle for the client session.

See Also

[InternetServer.ConnectEventArgs Class](#) | [SocketTools Namespace](#) | [OnConnect Event \(SocketTools.InternetServer\)](#)

InternetServer.ConnectEventArgs.Handle Property

Gets a value that specifies the socket handle for the client session.

[Visual Basic]

```
Public ReadOnly Property Handle As Integer
```

[C#]

```
public int Handle {get;}
```

Property Value

An integer value which specifies the client socket handle.

Remarks

The **Handle** property returns the socket handle for the client that generated the event. This handle can be used in conjunction with methods such as **Read** and **Write** to exchange data with the client.

See Also

[InternetServer.ConnectEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.OnDisconnect Event

Occurs when the remote host disconnects from the local system.

```
[Visual Basic]
Public Event OnDisconnect As OnDisconnectEventHandler

[C#]
public event OnDisconnectEventHandler OnDisconnect;
```

Event Data

The event handler receives an argument of type [InternetServer.DisconnectEventArgs](#) containing data related to this event. The following **InternetServer.DisconnectEventArgs** property provides information specific to this event.

Property	Description
Handle	Gets a value that specifies the socket handle for the client session.

Remarks

The **OnDisconnect** event is generated when the connection is terminated by the client and there is no more data available to be read.

It is not necessary to call the **Disconnect** method inside the **OnDisconnect** event handler because the client session is already in the process of disconnecting from the server.

User interface controls can only be accessed from the UI thread that created them, and attempting to update a control from another thread can result in the program becoming non-responsive or terminating abnormally. Because this event is generated in the context of the client thread, not the thread that created the class instance, you cannot directly modify a control from within this event handler. Instead, you must create a delegate and use the **Invoke** method to marshal invocations to the associated UI thread. For more information, refer to the documentation for the control.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.DisconnectEventArgs Class

Provides data for the OnDisconnect event.

For a list of all members of this type, see [InternetServer.DisconnectEventArgs Members](#).

System.Object

System.EventArgs

SocketTools.InternetServer.DisconnectEventArgs

[Visual Basic]

Public Class InternetServer.DisconnectEventArgs

Inherits EventArgs

[C#]

public class InternetServer.DisconnectEventArgs : EventArgs

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

DisconnectEventArgs specifies the socket handle for the current client session.

The [OnDisconnect](#) event is generated when the connection is terminated by the client and there is no more data available to be read. The **Handle** property specifies the socket handle of the client session which has terminated. It is important to note that the client handle is provided for informational purposes only and the application should not attempt to read or write data using this handle from within this event handler.

It is not necessary to call the **Disconnect** method inside the **OnDisconnect** event handler because the client session is already in the process of disconnecting from the server.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[InternetServer.DisconnectEventArgs Members](#) | [SocketTools Namespace](#) | [OnDisconnect Event \(SocketTools.InternetServer\)](#)

InternetServer.DisconnectEventArgs Members

[InternetServer.DisconnectEventArgs overview](#)

Public Instance Constructors

 InternetServer.DisconnectEventArgs Constructor	Initializes a new instance of the InternetServer.DisconnectEventArgs class.

Public Instance Properties

Handle	Gets a value that specifies the socket handle for the client session.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetServer.DisconnectEventArgs Class](#) | [SocketTools Namespace](#) | [OnDisconnect Event](#) ([SocketTools.InternetServer](#))

InternetServer.DisconnectEventArgs Constructor

Initializes a new instance of the [InternetServer.DisconnectEventArgs](#) class.

[Visual Basic]

```
Public Sub New()
```

[C#]

```
public InternetServer.DisconnectEventArgs();
```

See Also

[InternetServer.DisconnectEventArgs Class](#) | [SocketTools Namespace](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InternetServer.DisconnectEventArgs Properties

The properties of the **InternetServer.DisconnectEventArgs** class are listed below. For a complete list of **InternetServer.DisconnectEventArgs** class members, see the [InternetServer.DisconnectEventArgs Members](#) topic.

Public Instance Properties

Handle	Gets a value that specifies the socket handle for the client session.

See Also

[InternetServer.DisconnectEventArgs Class](#) | [SocketTools Namespace](#) | [OnDisconnect Event \(SocketTools.InternetServer\)](#)

InternetServer.DisconnectEventArgs.Handle Property

Gets a value that specifies the socket handle for the client session.

[Visual Basic]

```
Public ReadOnly Property Handle As Integer
```

[C#]

```
public int Handle {get;}
```

Property Value

An integer value which specifies the client socket handle.

Remarks

The **Handle** property returns the socket handle for the client that generated the event. This handle can be used in conjunction with methods such as **Read** and **Write** to exchange data with the client.

See Also

[InternetServer.DisconnectEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.OnError Event

Occurs when an socket operation fails.

```
[Visual Basic]
Public Event OnError As OnErrorHandler
```

```
[C#]
public event OnErrorHandler OnError;
```

Event Data

The event handler receives an argument of type [InternetServer.ErrorEventArgs](#) containing data related to this event. The following **InternetServer.ErrorEventArgs** properties provide information specific to this event.

Property	Description
Description	Gets a value which describes the last error that has occurred.
Error	Gets a value which specifies the last error that has occurred.
Handle	Gets a value that specifies the socket handle that generated the error.

Remarks

The **OnError** event occurs when a socket operation fails.

User interface controls can only be accessed from the UI thread that created them, and attempting to update a control from another thread can result in the program becoming non-responsive or terminating abnormally. Because this event may be generated in the context of the client or server thread, not the thread that created the class instance, you cannot directly modify a control from within this event handler. Instead, you must create a delegate and use the **Invoke** method to marshal invocations to the associated UI thread. For more information, refer to the documentation for the control.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.ErrorEventArgs Class

Provides data for the OnError event.

For a list of all members of this type, see [InternetServer.ErrorEventArgs Members](#).

System.Object

System.EventArgs

SocketTools.InternetServer.ErrorEventArgs

[Visual Basic]

Public Class InternetServer.ErrorEventArgs

Inherits EventArgs

[C#]

public class InternetServer.ErrorEventArgs : EventArgs

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

ErrorEventArgs specifies the numeric error code and a description of the error that has occurred.

An [OnError](#) event occurs when a method fails.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[InternetServer.ErrorEventArgs Members](#) | [SocketTools Namespace](#)

InternetServer.ErrorEventArgs Members

[InternetServer.ErrorEventArgs overview](#)

Public Instance Constructors

 InternetServer.ErrorEventArgs Constructor	Initializes a new instance of the InternetServer.ErrorEventArgs class.

Public Instance Properties

Description	Gets a value which describes the last error that has occurred.
Error	Gets a value which specifies the last error that has occurred.
Handle	Gets a value that specifies the socket handle that generated the error.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetServer.ErrorEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.ErrorEventArgs Constructor

Initializes a new instance of the [InternetServer.ErrorEventArgs](#) class.

[Visual Basic]

```
Public Sub New()
```

[C#]

```
public InternetServer.ErrorEventArgs();
```

See Also

[InternetServer.ErrorEventArgs Class](#) | [SocketTools Namespace](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InternetServer.ErrorEventArgs Properties

The properties of the **InternetServer.ErrorEventArgs** class are listed below. For a complete list of **InternetServer.ErrorEventArgs** class members, see the [InternetServer.ErrorEventArgs Members](#) topic.

Public Instance Properties

Description	Gets a value which describes the last error that has occurred.
Error	Gets a value which specifies the last error that has occurred.
Handle	Gets a value that specifies the socket handle that generated the error.

See Also

[InternetServer.ErrorEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.ErrorEventArgs.Description Property

Gets a value which describes the last error that has occurred.

[Visual Basic]

Public ReadOnly Property Description As String

[C#]

public string Description {get;}

Property Value

A string which describes the last error that has occurred.

See Also

[InternetServer.ErrorEventArgs Class](#) | [SocketTools Namespace](#) | [Error Property](#)

InternetServer.ErrorEventArgs.Error Property

Gets a value which specifies the last error that has occurred.

[Visual Basic]

Public **ReadOnly** **Property** **Error** As [ErrorCode](#)

[C#]

public [InternetServer.ErrorCode](#) **Error** {get;}

Property Value

[ErrorCode](#) enumeration which specifies the error.

See Also

[InternetServer.ErrorEventArgs Class](#) | [SocketTools Namespace](#) | [Description Property](#)

InternetServer.ErrorEventArgs.Handle Property

Gets a value that specifies the socket handle that generated the error.

[Visual Basic]

```
Public ReadOnly Property Handle As Integer
```

[C#]

```
public int Handle {get;}
```

Property Value

An integer value which specifies a socket handle.

Remarks

The **Handle** property returns the socket handle for the client or server that generated the event. If no server is active, then this property will return a value of -1.

See Also

[InternetServer.ErrorEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.OnRead Event

Occurs when data is available to be read from the client.

[Visual Basic]

Public Event OnRead As [OnReadEventHandler](#)

[C#]

public event [OnReadEventHandler](#) OnRead;

Event Data

The event handler receives an argument of type [InternetServer.ReadEventArgs](#) containing data related to this event. The following **InternetServer.ReadEventArgs** property provides information specific to this event.

Property	Description
Handle	Gets a value that specifies the socket handle for the client session.

Remarks

The **OnRead** event is generated when the client sends data to the server. The **Handle** event argument property specifies the handle to the client socket which can be used with the **Read** or **ReadLine** methods to read the data that was sent.

When this event fires, it guarantees that data can be read from the specified client without causing the current thread to enter a blocked state. However, calling this method multiple times inside the event handler may cause the current thread to block when there is no more data available to read. The **IsReadable** property can be used to determine if there is additional data available to be read.

User interface controls can only be accessed from the UI thread that created them, and attempting to update a control from another thread can result in the program becoming non-responsive or terminating abnormally. Because this event is generated in the context of the client thread, not the thread that created the class instance, you cannot directly modify a control from within this event handler. Instead, you must create a delegate and use the **Invoke** method to marshal invocations to the associated UI thread. For more information, refer to the documentation for the control.

Example

```
Private Sub Server_OnRead(ByVal sender As Object, ByVal e As System.EventArgs) Handles Socket.OnRead
    Dim strBuffer As String
    Dim nRead As Integer

    ' Read up to m_nBufferSize bytes of data from the client
    nRead = Server1.Read(e.Handle, strBuffer, m_nBufferSize)

    If nRead > 0 Then
        ' Process the data that has been read from the client
        ProcessData(strBuffer)
    End If
End Sub
```

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.ReadEventArgs Class

Provides data for the OnRead event.

For a list of all members of this type, see [InternetServer.ReadEventArgs Members](#).

System.Object

System.EventArgs

SocketTools.InternetServer.ReadEventArgs

[Visual Basic]

Public Class InternetServer.ReadEventArgs

Inherits EventArgs

[C#]

public class InternetServer.ReadEventArgs : EventArgs

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

ReadEventArgs specifies the socket handle for the current client session.

The [OnRead](#) event is generated when the client sends data to the server. The **Handle** event argument property specifies the handle to the client socket which can be used with the **Read** or **ReadLine** methods to read the data that was sent.

When this event fires, it guarantees that data can be read from the specified client without causing the current thread to enter a blocked state. However, calling this method multiple times inside the event handler may cause the current thread to block when there is no more data available to read. The **IsReadable** property can be used to determine if there is additional data available to be read.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[InternetServer.ReadEventArgs Members](#) | [SocketTools Namespace](#) | [OnRead Event](#)
([SocketTools.InternetServer](#))

InternetServer.ReadEventArgs Members

[InternetServer.ReadEventArgs overview](#)

Public Instance Constructors

 InternetServer.ReadEventArgs Constructor	Initializes a new instance of the InternetServer.ReadEventArgs class.

Public Instance Properties

Handle	Gets a value that specifies the socket handle for the client session.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object .
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object .

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object .

See Also

[InternetServer.ReadEventArgs Class](#) | [SocketTools Namespace](#) | [OnRead Event \(SocketTools.InternetServer\)](#)

InternetServer.ReadEventArgs Constructor

Initializes a new instance of the [InternetServer.ReadEventArgs](#) class.

[Visual Basic]

```
Public Sub New()
```

[C#]

```
public InternetServer.ReadEventArgs();
```

See Also

[InternetServer.ReadEventArgs Class](#) | [SocketTools Namespace](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InternetServer.ReadEventArgs Properties

The properties of the **InternetServer.ReadEventArgs** class are listed below. For a complete list of **InternetServer.ReadEventArgs** class members, see the [InternetServer.ReadEventArgs Members](#) topic.

Public Instance Properties

Handle	Gets a value that specifies the socket handle for the client session.

See Also

[InternetServer.ReadEventArgs Class](#) | [SocketTools Namespace](#) | [OnRead Event \(SocketTools.InternetServer\)](#)

InternetServer.ReadEventArgs.Handle Property

Gets a value that specifies the socket handle for the client session.

[Visual Basic]

```
Public ReadOnly Property Handle As Integer
```

[C#]

```
public int Handle {get;}
```

Property Value

An integer value which specifies the client socket handle.

Remarks

The **Handle** property returns the socket handle for the client that generated the event. This handle can be used in conjunction with methods such as **Read** and **Write** to exchange data with the client.

See Also

[InternetServer.ReadEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.OnTimeout Event

Occurs when a blocking operation fails to complete before the timeout period elapses.

[Visual Basic]

Public Event OnTimeout As OnTimeoutEventHandler

[C#]

public event OnTimeoutEventHandler OnTimeout;

Event Data

The event handler receives an argument of type [InternetServer.TimeoutEventArgs](#) containing data related to this event. The following **InternetServer.TimeoutEventArgs** property provides information specific to this event.

Property	Description
Handle	Gets a value that specifies the socket handle for the client session.

Remarks

The **OnTimeout** event occurs when a blocking operation, such as sending or receiving data on the socket, fails to complete before the specified timeout period elapses. The timeout period for a blocking operation can be adjusted by setting the **Timeout** property.

User interface controls can only be accessed from the UI thread that created them, and attempting to update a control from another thread can result in the program becoming non-responsive or terminating abnormally. Because this event is generated in the context of the client thread, not the thread that created the class instance, you cannot directly modify a control from within this event handler. Instead, you must create a delegate and use the **Invoke** method to marshal invocations to the associated UI thread. For more information, refer to the documentation for the control.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.TimeoutEventArgs Class

Provides data for the OnTimeout event.

For a list of all members of this type, see [InternetServer.TimeoutEventArgs Members](#).

System.Object

System.EventArgs

SocketTools.InternetServer.TimeoutEventArgs

[Visual Basic]

Public Class InternetServer.TimeoutEventArgs

Inherits EventArgs

[C#]

public class InternetServer.TimeoutEventArgs : EventArgs

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

TimeoutEventArgs specifies the socket handle for the current client session.

The [OnTimeout](#) event occurs when a blocking operation, such as sending or receiving data on the socket, fails to complete before the specified timeout period elapses. The timeout period for a blocking operation can be adjusted by setting the **Timeout** property.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[InternetServer.TimeoutEventArgs Members](#) | [SocketTools Namespace](#) | [OnTimeout Event \(SocketTools.InternetServer\)](#)

InternetServer.TimeoutEventArgs Members

[InternetServer.TimeoutEventArgs overview](#)

Public Instance Constructors

 InternetServer.TimeoutEventArgs Constructor	Initializes a new instance of the InternetServer.TimeoutEventArgs class.

Public Instance Properties

Handle	Gets a value that specifies the socket handle for the client session.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetServer.TimeoutEventArgs Class](#) | [SocketTools Namespace](#) | [OnTimeout Event](#)
([SocketTools.InternetServer](#))

InternetServer.TimeoutEventArgs Constructor

Initializes a new instance of the [InternetServer.TimeoutEventArgs](#) class.

[Visual Basic]

```
Public Sub New()
```

[C#]

```
public InternetServer.TimeoutEventArgs();
```

See Also

[InternetServer.TimeoutEventArgs Class](#) | [SocketTools Namespace](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InternetServer.TimeoutEventArgs Properties

The properties of the **InternetServer.TimeoutEventArgs** class are listed below. For a complete list of **InternetServer.TimeoutEventArgs** class members, see the [InternetServer.TimeoutEventArgs Members](#) topic.

Public Instance Properties

Handle	Gets a value that specifies the socket handle for the client session.

See Also

[InternetServer.TimeoutEventArgs Class](#) | [SocketTools Namespace](#) | [OnTimeout Event \(SocketTools.InternetServer\)](#)

InternetServer.TimeoutEventArgs.Handle Property

Gets a value that specifies the socket handle for the client session.

[Visual Basic]

```
Public ReadOnly Property Handle As Integer
```

[C#]

```
public int Handle {get;}
```

Property Value

An integer value which specifies the client socket handle.

Remarks

The **Handle** property returns the socket handle for the client that generated the event. This handle can be used in conjunction with methods such as **Read** and **Write** to exchange data with the client.

See Also

[InternetServer.TimeoutEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.OnWrite Event

Occurs when data can be written to the client.

```
[Visual Basic]
Public Event OnWrite As OnWriteEventHandler
```

```
[C#]
public event OnWriteEventHandler OnWrite;
```

Event Data

The event handler receives an argument of type [InternetServer.WriteEventArgs](#) containing data related to this event. The following **InternetServer.WriteEventArgs** property provides information specific to this event.

Property	Description
Handle	Gets a value that specifies the socket handle for the client session.

Remarks

The **OnWrite** event is generated when the client can accept data from the server. The **Handle** event argument property specifies the handle to the client socket and can be used in conjunction with the **Write** or **WriteLine** methods.

This event is typically fired once when the client connection is established with the server, the session thread starts and the client socket enters a writable state. If the internal send buffer for the client socket becomes full, this event will fire again when more data can be written to the socket. It is important to note that this event is level-triggered and will not fire repeatedly if the client socket is writable. Under most circumstances this event fire only once for each client session after the initial connection has been established.

User interface controls can only be accessed from the UI thread that created them, and attempting to update a control from another thread can result in the program becoming non-responsive or terminating abnormally. Because this event is generated in the context of the client thread, not the thread that created the class instance, you cannot directly modify a control from within this event handler. Instead, you must create a delegate and use the **Invoke** method to marshal invocations to the associated UI thread. For more information, refer to the documentation for the control.

See Also

[InternetServer Class](#) | [SocketTools Namespace](#)

InternetServer.WriteEventArgs Class

Provides data for the OnWrite event.

For a list of all members of this type, see [InternetServer.WriteEventArgs Members](#).

System.Object

System.EventArgs

SocketTools.InternetServer.WriteEventArgs

[Visual Basic]

Public Class InternetServer.WriteEventArgs

Inherits EventArgs

[C#]

public class InternetServer.WriteEventArgs : EventArgs

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

WriteEventArgs specifies the socket handle for the current client session.

The **OnWrite** event is generated when the client can accept data from the server. The **Handle** event argument property specifies the handle to the client socket and can be used in conjunction with the **Write** or **WriteLine** methods.

This event is typically fired once when the client connection is established with the server, the session thread starts and the client socket enters a writable state. If the internal send buffer for the client socket becomes full, this event will fire again when more data can be written to the socket. It is important to note that this event is level-triggered and will not fire repeatedly if the client socket is writable. Under most circumstances this event fire only once for each client session after the initial connection has been established.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[InternetServer.WriteEventArgs Members](#) | [SocketTools Namespace](#)

InternetServer.WriteEventArgs Members

[InternetServer.WriteEventArgs overview](#)

Public Instance Constructors

 InternetServer.WriteEventArgs Constructor	Initializes a new instance of the InternetServer.WriteEventArgs class.

Public Instance Properties

Handle	Gets a value that specifies the socket handle for the client session.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetServer.WriteEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.WriteEventArgs Constructor

Initializes a new instance of the [InternetServer.WriteEventArgs](#) class.

[Visual Basic]

```
Public Sub New()
```

[C#]

```
public InternetServer.WriteEventArgs();
```

See Also

[InternetServer.WriteEventArgs Class](#) | [SocketTools Namespace](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

InternetServer.WriteEventArgs Properties

The properties of the **InternetServer.WriteEventArgs** class are listed below. For a complete list of **InternetServer.WriteEventArgs** class members, see the [InternetServer.WriteEventArgs Members](#) topic.

Public Instance Properties

Handle	Gets a value that specifies the socket handle for the client session.

See Also

[InternetServer.WriteEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.WriteEventArgs.Handle Property

Gets a value that specifies the socket handle for the client session.

[Visual Basic]

```
Public ReadOnly Property Handle As Integer
```

[C#]

```
public int Handle {get;}
```

Property Value

An integer value which specifies the client socket handle.

Remarks

The **Handle** property returns the socket handle for the client that generated the event. This handle can be used in conjunction with methods such as **Read** and **Write** to exchange data with the client.

See Also

[InternetServer.WriteEventArgs Class](#) | [SocketTools Namespace](#)

InternetServer.OnAcceptEventHandler Delegate

Represents the method that will handle the [OnAccept](#) event.

[Visual Basic]

```
Public Delegate Sub InternetServer.OnAcceptEventHandler( _  
    ByVal sender As Object, _  
    ByVal e As AcceptEventArgs _  
)
```

[C#]

```
public delegate void InternetServer.OnAcceptEventHandler(  
    object sender,  
    AcceptEventArgs e  
);
```

Parameters

sender

The source of the event.

e

An [AcceptEventArgs](#) that contains the event data.

Remarks

When you create an **OnAcceptEventHandler** delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, until you remove the delegate.

Note that the declaration of your event handler must have the same parameters as the **OnAcceptEventHandler** delegate declaration.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[SocketTools Namespace](#) | [OnAccept Event](#)

InternetServer.OnCancelEventHandler Delegate

Represents the method that will handle the [OnCancel](#) event.

[Visual Basic]

```
Public Delegate Sub InternetServer.OnCancelEventHandler( _  
    ByVal sender As Object, _  
    ByVal e As CancelEventArgs _  
)
```

[C#]

```
public delegate void InternetServer.OnCancelEventHandler(  
    object sender,  
    CancelEventArgs e  
);
```

Parameters

sender

The source of the event.

e

An [CancelEventArgs](#) that contains the event data.

Remarks

When you create an **OnCancelEventHandler** delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, until you remove the delegate.

Note that the declaration of your event handler must have the same parameters as the **OnCancelEventHandler** delegate declaration.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[SocketTools Namespace](#)

InternetServer.OnConnectEventHandler Delegate

Represents the method that will handle the [OnConnect](#) event.

[Visual Basic]

```
Public Delegate Sub InternetServer.OnConnectEventHandler( _  
    ByVal sender As Object, _  
    ByVal e As ConnectEventArgs _  
)
```

[C#]

```
public delegate void InternetServer.OnConnectEventHandler(  
    object sender,  
    ConnectEventArgs e  
);
```

Parameters

sender

The source of the event.

e

An [ConnectEventArgs](#) that contains the event data.

Remarks

When you create an **OnConnectEventHandler** delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, until you remove the delegate.

Note that the declaration of your event handler must have the same parameters as the **OnConnectEventHandler** delegate declaration.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[SocketTools Namespace](#)

InternetServer.OnDisconnectEventHandler Delegate

Represents the method that will handle the [OnDisconnect](#) event.

[Visual Basic]

```
Public Delegate Sub InternetServer.OnDisconnectEventHandler( _  
    ByVal sender As Object, _  
    ByVal e As DisconnectEventArgs _  
)
```

[C#]

```
public delegate void InternetServer.OnDisconnectEventHandler(  
    object sender,  
    DisconnectEventArgs e  
);
```

Parameters

sender

The source of the event.

e

An [DisconnectEventArgs](#) that contains the event data.

Remarks

When you create an **OnDisconnectEventHandler** delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, until you remove the delegate.

Note that the declaration of your event handler must have the same parameters as the **OnDisconnectEventHandler** delegate declaration.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[SocketTools Namespace](#)

InternetServer.OnErrorHandler Delegate

Represents the method that will handle the [OnError](#) event.

[Visual Basic]

```
Public Delegate Sub InternetServer.OnErrorHandler( _  
    ByVal sender As Object, _  
    ByVal e As EventArgs _  
)
```

[C#]

```
public delegate void InternetServer.OnErrorHandler(  
    object sender,  
    EventArgs e  
);
```

Parameters

sender

The source of the event.

e

An [EventArgs](#) that contains the event data.

Remarks

When you create an **OnErrorHandler** delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, until you remove the delegate.

Note that the declaration of your event handler must have the same parameters as the **OnErrorHandler** delegate declaration.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[SocketTools Namespace](#)

InternetServer.OnReadEventHandler Delegate

Represents the method that will handle the [OnRead](#) event.

[Visual Basic]

```
Public Delegate Sub InternetServer.OnReadEventHandler( _  
    ByVal sender As Object, _  
    ByVal e As ReadEventArgs _  
)
```

[C#]

```
public delegate void InternetServer.OnReadEventHandler(  
    object sender,  
    ReadEventArgs e  
);
```

Parameters

sender

The source of the event.

e

An [ReadEventArgs](#) that contains the event data.

Remarks

When you create an **OnReadEventHandler** delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, until you remove the delegate.

Note that the declaration of your event handler must have the same parameters as the **OnReadEventHandler** delegate declaration.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[SocketTools Namespace](#)

InternetServer.OnTimeoutEventHandler Delegate

Represents the method that will handle the [OnTimeout](#) event.

[Visual Basic]

```
Public Delegate Sub InternetServer.OnTimeoutEventHandler( _  
    ByVal sender As Object, _  
    ByVal e As TimeoutEventArgs _  
)
```

[C#]

```
public delegate void InternetServer.OnTimeoutEventHandler(  
    object sender,  
    TimeoutEventArgs e  
);
```

Parameters

sender

The source of the event.

e

An [TimeoutEventArgs](#) that contains the event data.

Remarks

When you create an **OnTimeoutEventHandler** delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, until you remove the delegate.

Note that the declaration of your event handler must have the same parameters as the **OnTimeoutEventHandler** delegate declaration.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[SocketTools Namespace](#)

InternetServer.OnWriteEventHandler Delegate

Represents the method that will handle the [OnWrite](#) event.

[Visual Basic]

```
Public Delegate Sub InternetServer.OnWriteEventHandler( _  
    ByVal sender As Object, _  
    ByVal e As WriteEventArgs _  
)
```

[C#]

```
public delegate void InternetServer.OnWriteEventHandler(  
    object sender,  
    WriteEventArgs e  
);
```

Parameters

sender

The source of the event.

e

An [WriteEventArgs](#) that contains the event data.

Remarks

When you create an **OnWriteEventHandler** delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, until you remove the delegate.

Note that the declaration of your event handler must have the same parameters as the **OnWriteEventHandler** delegate declaration.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[SocketTools Namespace](#)

InternetServer.ErrorCode Enumeration

The error codes used by the class.

[Visual Basic]

```
Public Enum InternetServer.ErrorCode
```

[C#]

```
public enum InternetServer.ErrorCode
```

Remarks

The **ErrorCode** enumeration specifies what error has occurred when a method fails. The current error code may be determined by checking the value of the **LastError** property.

Note that the last error code is only meaningful if the previous operation has failed.

Members

Member Name	Description
None	No error.
NotHandleOwner	Handle not owned by the current thread.
OperationCanceled	The blocking operation has been canceled.
OutOfMemory	Out of memory.
AccessDenied	Access denied.
InvalidParameter	Invalid argument passed to function.
InvalidBuffer	Invalid memory address passed to function.
BufferTooSmall	The specified buffer is not large enough to contain the data.
NoHandles	No more handles are available to this process.
OperationWouldBlock	The specified operation would block the current thread.
OperationInProgress	A blocking operation is currently in progress.
AlreadyInProgress	The specified operation is already in progress.
InvalidHandle	Invalid handle passed to function.
InvalidAddress	Invalid network address specified.
InvalidSize	Datagram is too large to fit in specified buffer.
InvalidProtocol	Invalid network protocol specified.
ProtocolNotAvailable	The specified network protocol is not available.
ProtocolNotSupported	The specified protocol is not supported.
SocketNotSupported	The specified socket type is not supported.
InvalidOption	The specified option is invalid.
ProtocolFamily	Specified protocol family is not supported.

ProtocolAddress	The specified address is invalid for this protocol family.
AddressInUse	The specified address is in use by another process.
AddressUnavailable	The specified address cannot be assigned.
NetworkUnavailable	The networking subsystem is unavailable.
NetworkUnreachable	The specified network is unreachable.
NetworkReset	Network dropped connection on remote reset.
ConnectionAborted	Connection was aborted due to timeout or other failure.
ConnectionReset	Connection was reset by remote network.
OutOfBuffers	No buffer space is available.
AlreadyConnected	Connection already established with remote host.
NotConnected	No connection established with remote host.
ConnectionShutdown	Unable to send or receive data after connection shutdown.
OperationTimeout	The specified operation has timed out.
ConnectionRefused	The connection has been refused by the remote host.
HostUnavailable	The specified host is unavailable.
HostUnreachable	Remote host is unreachable.
TooManyProcesses	Too many processes are using the networking subsystem.
TooManyThreads	Too many threads have been created by the current process.
TooManySessions	Too many client sessions have been created by the current process.
InternalFailure	An unexpected internal error has occurred.
NetworkNotReady	Network subsystem is not ready for communication.
InvalidVersion	This version of the operating system is not supported.
NetworkNotInitialized	The networking subsystem has not been initialized.
RemoteShutdown	The remote host has initiated a graceful shutdown sequence.
InvalidHostName	The specified hostname is invalid or could not be resolved.
HostNameNotFound	The specified hostname could not be found.
HostNameRefused	Unable to resolve hostname, request refused.
HostNameNotResolved	Unable to resolve hostname, no address for specified host.

InvalidLicense	The license for this product is invalid.
ProductNotLicensed	This product is not licensed to perform this operation.
NotImplemented	This function has not been implemented on this platform.
UnknownLocalhost	Unable to determine local host name.
InvalidHostAddress	Invalid host address specified.
InvalidServicePort	Invalid service port number specified.
InvalidServiceName	Invalid or unknown service name specified.
InvalidEventId	Invalid event identifier specified.
OperationNotBlocking	No blocking operation in progress on this socket.
SecurityNotInitialized	Unable to initialize security interface for this process.
SecurityContext	Unable to establish security context for this session.
SecurityCredentials	Unable to open certificate store or establish security credentials.
SecurityCertificate	Unable to validate the certificate chain for this session.
SecurityDecryption	Unable to decrypt data stream.
SecurityEncryption	Unable to encrypt data stream.
OperationNotSupported	The specified operation is not supported.
InvalidProtocolVersion	Invalid application protocol version specified.
NetworkAdapter	Unable to determine network adapter configuration.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[SocketTools Namespace](#) | [LastError Property](#) | [LastErrorString Property](#) | [OnError Event](#)

InternetServer.SecurityProtocols Enumeration

The supported security protocols.

This enumeration has a FlagsAttribute attribute that allows a bitwise combination of its member values.

[Visual Basic]

<Flags>

Public Enum InternetServer.SecurityProtocols

[C#]

[Flags]

public enum InternetServer.SecurityProtocols

Remarks

The **SecurityProtocols** enumeration specifies one or more security protocols to be used when accepting a connection from a client. Multiple protocols may be specified if necessary and the actual protocol used will be negotiated with the client. It is recommended that most servers use the **Default** protocol when accepting secure connections.

Members

Member Name	Description	Value
None	No security protocol will be used and the connection will not be secure.	0
Ssl2	Obsolete. The SSL 2.0 protocol has been selected. This protocol has been deprecated and is no longer widely used. It is not recommended that this protocol be used when accepting secure connections.	1
Ssl3	Obsolete. The SSL 3.0 protocol has been selected. This protocol has been deprecated and is no longer widely used. It is not recommended that this protocol be used when accepting secure connections. In most cases, this protocol is only selected if TLS is not supported by the client.	2
Tls10	Obsolete. The TLS 1.0 protocol has been selected. This version of the protocol is commonly used by older clients and is the only version of TLS supported on Windows platforms prior to Windows 7 SP1 and Windows Server 2008 R2. This version has been deprecated and should not be used.	4
Tls11	Obsolete. The TLS 1.1 protocol has been selected. This version of TLS is supported on Windows 7 SP1 and Windows Server	8

	2008 R2 and later versions of the operating system. This version has been deprecated and should not be used.	
Tls12	The TLS 1.2 protocol has been selected. This is the default version of the protocol and is supported on Windows 7 SP1 and Windows Server 2008 R2 and later versions of Windows. It is recommended that you use this version of TLS.	16
Tls13	The TLS 1.3 protocol has been selected. This is the latest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later. If this protocol version is not supported, TLS 1.2 will be used instead.	32
Default	The default selection of security protocols will be used when accepting a connection. The TLS 1.3 or TLS 1.2 protocols will be negotiated with the client, in that order of preference. This option will always request the latest version of the preferred security protocols and is the recommended value.	48
Unknown	An unknown or unsupported security protocol has been specified. This value indicates an error condition.	4096

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[SocketTools Namespace](#)

InternetServer.ServerOptions Enumeration

The supported server options used when accepting a client connection.

This enumeration has a `FlagsAttribute` attribute that allows a bitwise combination of its member values.

[Visual Basic]

<Flags>

Public Enum InternetServer.ServerOptions

[C#]

[Flags]

public enum InternetServer.ServerOptions

Members

Member Name	Description	Value
None	No option specified.	0
DontRoute	This option specifies default routing should not be used. This option should not be specified unless absolutely necessary.	2
KeepAlive	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This option is only valid for stream sockets.	4
ReuseAddress	This option specifies the local address can be reused. This option is commonly used by server applications.	8
NoDelay	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.	16
Secure	This option specifies that a secure, encrypted connection will be established with the remote host.	4096
SecureFallback	This option specifies the server should permit the use of less secure cipher suites for compatibility with legacy clients. If this option is specified, the server will permit connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.	32768

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

InternetServer.ServerPriority Enumeration

The supported server priorities.

This enumeration has a FlagsAttribute attribute that allows a bitwise combination of its member values.

[Visual Basic]

<Flags>

Public Enum InternetServer.ServerPriority

[C#]

[Flags]

public enum InternetServer.ServerPriority

Members

Member Name	Description	Value
Background	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. Each client thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.	0
Low	This priority lowers the overall resource utilization for the client session and meters the processor utilization for the client session. Each client thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.	1
Normal	The default priority which balances resource and processor utilization. It is recommended that most applications use this priority.	2
High	This priority increases the overall resource utilization for each client session and their threads will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.	3
Critical	This priority can significantly increase processor, memory and network utilization. Each client thread will be given higher scheduling priority and will be more responsive to network events. It is not recommended that this priority be used on a system with a single processor.	4

Invalid	An invalid transfer priority which indicates an error condition.	-1
Default	The default server priority. This is the same as specifying Normal .	2
Lowest	The lowest valid server priority. This is the same as specifying Background .	0
Highest	The highest valid server priority. This is the same as specifying Critical .	4

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[SocketTools Namespace](#)

InternetServer.ServerStatus Enumeration

The current state of the server.

```
[Visual Basic]
Public Enum InternetServer.ServerStatus
```

```
[C#]
public enum InternetServer.ServerStatus
```

Members

Member Name	Description
Inactive	The server is currently inactive. This status is returned when no server has been started, or after a server has been stopped.
Started	The server has initialized and is preparing to listen for client connections. This status is returned after the server thread has been started, but before the listening socket has been created.
Listening	The server is actively listening for incoming client connections. This status is returned after the server thread has been started and the listening socket has been created.
Suspended	The server has been suspended and is no longer accepting client connections. Any incoming client connection is queued, and will be accepted when the server resumes normal operation.
Shutdown	The server has been stopped and is in the process of terminating all active client sessions.

Requirements

- Namespace:** [SocketTools](#)
- Assembly:** SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[SocketTools Namespace](#)

InternetServer.TraceOptions Enumeration

The supported trace logging options.

This enumeration has a `FlagsAttribute` attribute that allows a bitwise combination of its member values.

[Visual Basic]

<Flags>

Public Enum InternetServer.TraceOptions

[C#]

[Flags]

public enum InternetServer.TraceOptions

Remarks

The **TraceOptions** enumeration determines what kind of information is written to the logfile specified by the **TraceFile** property. These options are only meaningful when logging is enabled by setting the **Trace** property to **true**.

Members

Member Name	Description	Value
Info	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.	0
Error	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.	1
Warning	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.	2
HexDump	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload	4

	contents and responses.	
Truncate	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.	8
Process	All network function calls made by the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.	4096
Default	The same as specifying the HexDump and Process options.	4100

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[SocketTools Namespace](#)

InternetServer.RuntimeLicenseAttribute Class

Attribute that defines the runtime license key for the class.

For a list of all members of this type, see [InternetServer.RuntimeLicenseAttribute Members](#).

System.Object

System.Attribute

SocketTools.InternetServer.RuntimeLicenseAttribute

[Visual Basic]

```
<AttributeUsage(ValidOn:=AttributeTargets.Assembly, AllowMultiple:=False,
    Inherited:=True)>
Public Class InternetServer.RuntimeLicenseAttribute
    Inherits Attribute
```

[C#]

```
[AttributeUsage(ValidOn=AttributeTargets.Assembly, AllowMultiple=False,
    Inherited=True)]
public class InternetServer.RuntimeLicenseAttribute : Attribute
```

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

The RuntimeLicense attribute is used to define the runtime license key that will be used when an instance of the class is created. This attribute is defined in the assembly information module for the language, such as AssemblyInfo.cs when programming C#. The runtime license key must be defined if you wish to redistribute your application.

The runtime license key can be generated using the License Manager utility that is included with the product. Note that if you have installed an evaluation license, you will not have a runtime license key and cannot redistribute any applications which use the InternetServer class.

Example

```
<Assembly: SocketTools.InternetServer.RuntimeLicense("abcdefghijklmnop")>
```

```
[assembly: SocketTools.InternetServer.RuntimeLicense("abcdefghijklmnop")]
```

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[InternetServer.RuntimeLicenseAttribute Members](#) | [SocketTools Namespace](#)

InternetServer.RuntimeLicenseAttribute Members

[InternetServer.RuntimeLicenseAttribute overview](#)

Public Instance Constructors

 InternetServer.RuntimeLicenseAttribute Constructor	Constructor for the RuntimeLicense attribute which defines the runtime license key.

Public Instance Properties

LicenseKey	Returns the value of the runtime license key.
TypeId (inherited from Attribute)	When implemented in a derived class, gets a unique identifier for this Attribute.

Public Instance Methods

Equals (inherited from Attribute)	
GetHashCode (inherited from Attribute)	Returns the hash code for this instance.
GetType (inherited from Object)	Gets the Type of the current instance.
IsDefaultAttribute (inherited from Attribute)	When overridden in a derived class, returns an indication whether the value of this instance is the default value for the derived class.
Match (inherited from Attribute)	When overridden in a derived class, returns a value indicating whether this instance equals a specified object.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetServer.RuntimeLicenseAttribute Class](#) | [SocketTools Namespace](#)

InternetServer.RuntimeLicenseAttribute Constructor

Constructor for the RuntimeLicense attribute which defines the runtime license key.

[Visual Basic]

```
Public Sub New( _  
    ByVal licenseKey As String _  
)
```

[C#]

```
public InternetServer.RuntimeLicenseAttribute(  
    string licenseKey  
);
```

Parameters

licenseKey

A string argument which specifies the runtime license key which will be used to initialize the class library.

Remarks

The runtime license key can be generated using the License Manager utility that is included with the product. Note that if you have installed an evaluation license, you will not have a runtime license key and cannot redistribute any applications which use the InternetServer class.

See Also

[InternetServer.RuntimeLicenseAttribute Class](#) | [SocketTools Namespace](#)

InternetServer.RuntimeLicenseAttribute Properties

The properties of the **InternetServer.RuntimeLicenseAttribute** class are listed below. For a complete list of **InternetServer.RuntimeLicenseAttribute** class members, see the [InternetServer.RuntimeLicenseAttribute Members](#) topic.

Public Instance Properties

LicenseKey	Returns the value of the runtime license key.
TypeId (inherited from Attribute)	When implemented in a derived class, gets a unique identifier for this Attribute.

See Also

[InternetServer.RuntimeLicenseAttribute Class](#) | [SocketTools Namespace](#)

InternetServer.RuntimeLicenseAttribute.LicenseKey Property

Returns the value of the runtime license key.

[Visual Basic]

Public Property LicenseKey As String

[C#]

public string LicenseKey {get; set;}

Property Value

A string which contains the runtime license key.

See Also

[InternetServer.RuntimeLicenseAttribute Class](#) | [SocketTools Namespace](#)

InternetServerException Class

The exception that is thrown when a socket error occurs.

For a list of all members of this type, see [InternetServerException Members](#).

System.Object

System.Exception

System.ApplicationException

SocketTools.InternetServerException

[Visual Basic]

Public Class InternetServerException

Inherits ApplicationException

[C#]

public class InternetServerException : ApplicationException

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

A InternetServerException is thrown by the InternetServer class when an error occurs.

The default constructor for the InternetServerException class sets the **ErrorCode** property to the last socket error that occurred.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.InternetServer (in SocketTools.InternetServer.dll)

See Also

[InternetServerException Members](#) | [SocketTools Namespace](#)

InternetServerException Members

[InternetServerException overview](#)

Public Instance Constructors

 InternetServerException	Overloaded. Initializes a new instance of the InternetServerException class.

Public Instance Properties

ErrorCode	Gets a value which specifies the error that caused the exception.
Handle	Gets a value which specifies the socket handle that generated the error.
HelpLink (inherited from Exception)	Gets or sets a link to the help file associated with this exception.
InnerException (inherited from Exception)	Gets the Exception instance that caused the current exception.
Message	Gets a value which describes the error that caused the exception.
Number	Gets a value which specifies the numeric value of the error that caused the exception.
Source (inherited from Exception)	Gets or sets the name of the application or the object that causes the error.
StackTrace (inherited from Exception)	Gets a string representation of the frames on the call stack at the time the current exception was thrown.
TargetSite (inherited from Exception)	Gets the method that throws the current exception.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetBaseException (inherited from Exception)	When overridden in a derived class, returns the Exception that is the root cause of one or more subsequent exceptions.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetObjectData (inherited from Exception)	When overridden in a derived class, sets the SerializationInfo with information about the exception.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString	Creates and returns a string representation of the current exception.

Protected Instance Properties

 HRESULT (inherited from Exception)	Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetServerException Class](#) | [SocketTools Namespace](#)

InternetServerException Constructor

Initializes a new instance of the [InternetServerException](#) class with the last network error code.

Overload List

Initializes a new instance of the [InternetServerException](#) class with the last network error code.

```
public InternetServerException();
```

Initializes a new instance of the [InternetServerException](#) class with the last network error code for the specified socket.

```
public InternetServerException(int);
```

Initializes a new instance of the [InternetServerException](#) class with a specified error number.

```
public InternetServerException(int,int);
```

Initializes a new instance of the [InternetServerException](#) class with a specified error message.

```
public InternetServerException(int,string);
```

Initializes a new instance of the [InternetServerException](#) class with a specified error message and a reference to the inner exception that is the cause of this exception.

```
public InternetServerException(int,string,Exception);
```

See Also

[InternetServerException Class](#) | [SocketTools Namespace](#)

InternetServerException Constructor ()

Initializes a new instance of the [InternetServerException](#) class with the last network error code.

[Visual Basic]

Overloads Public Sub New()

[C#]

public InternetServerException();

Remarks

The ctor constructor sets the **ErrorCode** property to the last socket error that occurred. For more information about the errors that may occur, refer to the [InternetServer.ErrorCode](#) enumeration.

See Also

[InternetServerException Class](#) | [SocketTools Namespace](#) | [InternetServerException Constructor Overload List](#)

InternetServerException Constructor (Int32)

Initializes a new instance of the [InternetServerException](#) class with the last network error code for the specified socket.

[Visual Basic]

```
Overloads Public Sub New( _  
    ByVal handle As Integer _  
)
```

[C#]

```
public InternetServerException(  
    int handle  
);
```

Parameters

handle

An integer value which specifies the handle to the socket which generated the error.

Remarks

The ctor constructor sets the **ErrorCode** property to the last socket error that occurred. For more information about the errors that may occur, refer to the [InternetServer.ErrorCode](#) enumeration.

See Also

[InternetServerException Class](#) | [SocketTools Namespace](#) | [InternetServerException Constructor Overload List](#)

InternetServerException Constructor (Int32, String)

Initializes a new instance of the [InternetServerException](#) class with a specified error message.

[Visual Basic]

```
Overloads Public Sub New( _  
    ByVal handle As Integer, _  
    ByVal message As String _  
)
```

[C#]

```
public InternetServerException(  
    int handle,  
    string message  
);
```

Parameters

handle

An integer value which specifies the handle to the socket which generated the error.

message

The error message that explains the reason for the exception.

Remarks

The content of the *message* parameter is intended to be understood by humans. The caller of this constructor is required to ensure that this string has been localized for the current system culture.

See Also

[InternetServerException Class](#) | [SocketTools Namespace](#) | [InternetServerException Constructor Overload List](#)

InternetServerException Constructor (Int32, String, Exception)

Initializes a new instance of the [InternetServerException](#) class with a specified error message and a reference to the inner exception that is the cause of this exception.

[Visual Basic]

```
Overloads Public Sub New( _  
    ByVal handle As Integer, _  
    ByVal message As String, _  
    ByVal innerException As Exception _  
)
```

[C#]

```
public InternetServerException(  
    int handle,  
    string message,  
    Exception innerException  
);
```

Parameters

handle

An integer value which specifies the handle to the socket which generated the error.

message

The error message that explains the reason for the exception.

innerException

The exception that is the cause of the current exception. If the *innerException* parameter is not a null reference, the current exception is raised in a **catch** block that handles the inner exception.

Remarks

The content of the message parameter is intended to be understood by humans. The caller of this constructor is required to ensure that this string has been localized for the current system culture.

An exception that is thrown as a direct result of a previous exception should include a reference to the previous exception in the **InnerException** property. The **InnerException** property returns the same value that is passed into the constructor, or a null reference if the **InnerException** property does not supply the inner exception value to the constructor.

See Also

[InternetServerException Class](#) | [SocketTools Namespace](#) | [InternetServerException Constructor Overload List](#)

InternetServerException Properties

The properties of the **InternetServerException** class are listed below. For a complete list of **InternetServerException** class members, see the [InternetServerException Members](#) topic.

Public Instance Properties

ErrorCode	Gets a value which specifies the error that caused the exception.
Handle	Gets a value which specifies the socket handle that generated the error.
HelpLink (inherited from Exception)	Gets or sets a link to the help file associated with this exception.
InnerException (inherited from Exception)	Gets the Exception instance that caused the current exception.
Message	Gets a value which describes the error that caused the exception.
Number	Gets a value which specifies the numeric value of the error that caused the exception.
Source (inherited from Exception)	Gets or sets the name of the application or the object that causes the error.
StackTrace (inherited from Exception)	Gets a string representation of the frames on the call stack at the time the current exception was thrown.
TargetSite (inherited from Exception)	Gets the method that throws the current exception.

Protected Instance Properties

 HRESULT (inherited from Exception)	Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

See Also

[InternetServerException Class](#) | [SocketTools Namespace](#)

InternetServerException.ErrorCode Property

Gets a value which specifies the error that caused the exception.

[Visual Basic]

```
Public ReadOnly Property ErrorCode As ErrorCode
```

[C#]

```
public InternetServer.ErrorCode ErrorCode {get;}
```

Property Value

Returns a [InternetServer.ErrorCode](#) enumeration value which specifies the error code.

Remarks

The **ErrorCode** property returns the error code that specifies the cause of the exception.

The default constructor for the [InternetServerException](#) class sets the error code to the last network error that occurred. For more information about the errors that may occur, refer to the [InternetServer.ErrorCode](#) enumeration.

See Also

[InternetServerException Class](#) | [SocketTools Namespace](#)

InternetServerException.Message Property

Gets a value which describes the error that caused the exception.

[Visual Basic]

Overrides Public ReadOnly Property Message As String

[C#]

public override string Message {get;}

Property Value

A string which describes the error that caused the exception.

Remarks

The **Message** property returns a string which describes the error that caused the exception.

See Also

[InternetServerException Class](#) | [SocketTools Namespace](#)

InternetServerException.Number Property

Gets a value which specifies the numeric value of the error that caused the exception.

[Visual Basic]

```
Public ReadOnly Property Number As Integer
```

[C#]

```
public int Number {get;}
```

Property Value

An integer value that specifies the error that caused the exception.

Remarks

The **Number** property returns an integer value which specifies the numeric value of the error that caused the exception. This value may be the same as values returned by the Windows Sockets API. For more information about socket error codes, see the Windows Socket Version 2 API error code documentation in MSDN.

See Also

[InternetServerException Class](#) | [SocketTools Namespace](#)

InternetServerException Methods

The methods of the **InternetServerException** class are listed below. For a complete list of **InternetServerException** class members, see the [InternetServerException Members](#) topic.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetBaseException (inherited from Exception)	When overridden in a derived class, returns the Exception that is the root cause of one or more subsequent exceptions.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetObjectData (inherited from Exception)	When overridden in a derived class, sets the SerializationInfo with information about the exception.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString	Creates and returns a string representation of the current exception.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[InternetServerException Class](#) | [SocketTools Namespace](#)

InternetServerException.ToString Method

Creates and returns a string representation of the current exception.

[Visual Basic]

Overrides Public Function ToString() As String

[C#]

public override string ToString();

Return Value

A string representation of the current exception.

Remarks

The **ToString** method returns a representation of the current exception that is intended to be understood by humans. Where the exception contains culture-sensitive data, the string representation returned by **ToString** is required to take into account the current system culture. Although there are no exact requirements for the format of the returned string, it should attempt to reflect the value of the object as perceived by the user.

This implementation of **ToString** obtains the numeric error code value and a description of the error that caused the current exception. If there is no error message or it is an empty string, then no error message is returned.

This method overrides **ApplicationException.ToString**.

See Also

[InternetServerException Class](#) | [SocketTools Namespace](#)

SocketWrench Class

A general purpose TCP/IP networking class for developing client and server applications.

For a list of all members of this type, see [SocketWrench Members](#).

System.Object

SocketTools.SocketWrench

[Visual Basic]

Public Class SocketWrench
Implements IDisposable

[C#]

public class SocketWrench : IDisposable

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

At the core of each of the SocketTools networking classes is the Windows Sockets API. This provides a low level interface for sending and receiving data over the Internet or a local intranet using the Transmission Control Protocol (TCP) and/or User Datagram Protocol (UDP). The SocketWrench class provides a simpler interface to the Windows Sockets API, without sacrificing features or functionality. Using SocketWrench, you can easily create client and server applications while avoiding many of the mundane tasks and common problems that programmers face when developing Internet applications.

This class supports secure connections using the Transport Layer Security (TLS) protocols and can also be used to create secure, custom server programs. Both implicit and explicit TLS connections are supported, enabling the class to work with a wide variety of client and server applications without requiring that you use third-party classes or understand Microsoft's cryptography classes.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketWrench Members](#) | [SocketTools Namespace](#)

SocketWrench Members

[SocketWrench overview](#)

Public Instance Constructors

 SocketWrench Constructor	Initializes a new instance of the SocketWrench class.

Public Instance Fields

AdapterAddress	Returns the IP address associated with the specified network adapter.
HostAlias	Returns the aliases for a given host name.

Public Instance Properties

AdapterCount	Get the number of available local and remote network adapters.
AddressFamily	Gets and sets a value that determines which version of the Internet Protocol will be used.
AtMark	Get a value that indicates if the next receive will return urgent data.
AutoResolve	Gets and sets a value that determines if host names and addresses are automatically resolved.
Backlog	Gets and sets a value that indicates the number of connections that may be queued for a listening socket.
Blocking	Gets and sets a value which indicates if the socket is in blocking mode.
Broadcast	Gets and sets a value which indicates if datagrams will be broadcast over the local network.
ByteOrder	Gets and sets a value which indicates how integer data is read and written to the socket.
CertificateExpires	Get a value that specifies the date that the security certificate expires.
CertificateIssued	Get a value that specifies the date that the security certificate was issued.
CertificateIssuer	Get a value that provides information about the organization that issued the certificate.
CertificateName	Gets and sets a value that specifies the name of the security certificate.
CertificatePassword	Gets and sets the password associated with the security certificate.
CertificateStatus	Gets a value which indicates the status of the security certificate returned by the remote host.
CertificateStore	Gets and sets a value that specifies the name of the

	local certificate store.
CertificateSubject	Gets a value that provides information about the organization that the server certificate was issued to.
CertificateUser	Gets and sets the user that owns the security certificate.
CipherStrength	Gets a value that indicates the length of the key used by the encryption algorithm for a secure connection.
CodePage	Gets and sets the code page used when reading and writing text.
ExternalAddress	Gets a value that specifies the external Internet address for the local system.
Handle	Gets a value that specifies the socket handle allocated for the current session.
HashStrength	Gets a value which specifies the length of the message digest that was selected for a secure connection.
HostAddress	Gets and sets a value which specifies the Internet address used to establish a connection.
HostAliases	Returns the number of aliases for the current host name.
HostFile	Gets and sets a value that specifies the name of a host file used to resolve host names and addresses.
HostName	Gets and sets a value which specifies the host name used to establish a connection.
InLine	Gets and sets a value that indicates if urgent data is received in-line with non-urgent data.
IsBlocked	Gets a value which indicates if the current thread is performing a blocking socket operation.
IsClosed	Gets a value which indicates if the connection to the remote host has been closed.
IsConnected	Gets a value which indicates if a connection to the remote host has been established.
IsInitialized	Gets a value which indicates if the current instance of the class has been initialized successfully.
IsListening	Gets a value which indicates if the socket is listening for client connections.
IsReadable	Gets a value which indicates if there is data available to be read from the socket.
IsWritable	Gets a value which indicates if data can be written to the socket without blocking.
KeepAlive	Gets and sets a value which indicates if keep-alive packets are sent on a connected socket.

LastError	Gets and sets a value which specifies the last error that has occurred.
LastErrorString	Gets a value which describes the last error that has occurred.
Linger	Gets and sets a value which specifies the number of seconds to wait for the socket to disconnect from the remote host.
LocalAddress	Gets and sets the local Internet address that the socket will be bound to.
LocalName	Gets a value which specifies the host name for the local system.
LocalPort	Gets and sets a value which specifies the local port number the socket will be bound to.
LocalService	Gets and sets a value which specifies the local service the socket will be bound to.
NoDelay	Gets and sets a value which specifies if the Nagle algorithm should be enabled or disabled.
Options	Gets and sets a value which specifies one or more socket options.
PeerAddress	Gets a value that specifies the Internet address of the remote host.
PeerName	Gets a value that specifies the name of the remote host.
PeerPort	Gets a value that specifies the port number used by the remote host.
PhysicalAddress	Gets a value which specifies the MAC address for the local system's network adapter.
Protocol	Gets and sets a value which specifies the socket protocol.
RemotePort	Gets and sets a value which specifies the remote port number.
RemoteService	Gets and sets a value which specifies the remote service.
ReservedPort	Gets and sets a value which indicates if a reserved port number was used.
ReuseAddress	Gets and sets a value which indicates if a socket address can be reused.
Route	Gets and sets a value which indicates if packets should be routed.
Secure	Gets and sets a value which specifies if a secure connection is established.
SecureCipher	Gets a value that specifies the encryption algorithm

	used for a secure connection.
SecureHash	Gets a value that specifies the message digest algorithm used for a secure connection.
SecureKeyExchange	Gets a value that specifies the key exchange algorithm used for a secure connection.
SecureProtocol	Gets and sets a value which specifies the protocol used for a secure connection.
Status	Gets a value which specifies the current status of the socket.
ThreadModel	Gets and sets a value which specifies the threading model for the class instance.
ThrowError	Gets and sets a value which specifies if method calls should throw exceptions when an error occurs.
Timeout	Gets and sets a value which specifies a timeout period in seconds.
Trace	Gets and sets a value which indicates if network function logging is enabled.
TraceFile	Gets and sets a value which specifies the name of the network function tracing logfile.
TraceFlags	Gets and sets a value which specifies the network function tracing flags.
Urgent	Gets and sets a value which specifies if urgent data will be read or written.
Version	Gets a value which returns the current version of the SocketWrench class library.
VpnAdapter	Gets the name of the active Virtual Private Network adapter.
VpnAddress	Gets the IP address of the active Virtual Private Network adapter.

Public Instance Methods

Abort	Abort the connection with a remote host.
Accept	Overloaded. Accepts a client connection on a listening socket, specifying a timeout period and one or more socket options.
AttachThread	Attach an instance of the class to the current thread
Bind	Overloaded. Bind the socket to the specified local address and port number.
Cancel	Cancel the current blocking socket operation.
Connect	Overloaded. Establish a connection with a remote host.
ConnectUrl	Overloaded. Establish a connection with a remote

	host using a URL.
Disconnect	Terminate the connection with a remote host.
Dispose	Overloaded. Releases all resources used by SocketWrench .
Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
Flush	Flush the contents of the send and receive socket buffers.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
Initialize	Overloaded. Initialize an instance of the SocketWrench class.
Listen	Overloaded. Listen for incoming client connections, specifying the local network address, port number and connection backlog.
Peek	Overloaded. Read data from the socket and store it in a byte array, but do not remove the data from the socket buffers.
Read	Overloaded. Read data from the socket and store it in a byte array.
ReadFrom	Overloaded. Read data from the socket and store it in a byte array.
ReadLine	Overloaded. Read up to a line of data from the socket and return it in a string buffer.
ReadStream	Overloaded. Read a data stream from the socket and store it in the specified byte array.
Reject	Rejects a connection request from a remote host.
Reset	Reset the internal state of the object, resetting all properties to their default values.
Resolve	Resolves a host name to a host IP address.
Shutdown	Overloaded. Disable sending or receiving data on the socket.
StoreStream	Overloaded. Reads a data stream from the socket and stores it in the specified file.
ToString (inherited from Object)	Returns a String that represents the current Object.
Uninitialize	Uninitialize the class library and release any resources allocated for the current thread.
Write	Overloaded. Write one or more bytes of data to the socket.
WriteLine	Overloaded. Send a line of text to the remote host,

	terminated by a carriage-return and linefeed.
WriteStream	Overloaded. Write a stream of bytes to the socket.
WriteTo	Overloaded. Write one or more bytes of data to the socket.

Public Instance Events

OnAccept	Occurs when a remote host attempts to establish a connection with the local system.
OnCancel	Occurs when a blocking socket operation is canceled.
OnConnect	Occurs when a connection is established with the remote host.
OnDisconnect	Occurs when the remote host disconnects from the local system.
OnError	Occurs when an socket operation fails.
OnProgress	Occurs as a data stream is being read or written to the socket.
OnRead	Occurs when data is available to be read from the socket.
OnTimeout	Occurs when a blocking operation fails to complete before the timeout period elapses.
OnWrite	Occurs when data can be written to the socket.

Protected Instance Methods

 Dispose	Overloaded. Releases the unmanaged resources allocated by the SocketWrench class and optionally releases the managed resources.
 Finalize	Destroys an instance of the class, releasing the resources allocated for the session and unloading the networking library.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench Constructor

Initializes a new instance of the SocketWrench class.

```
[Visual Basic]
Public Sub New()
```

```
[C#]
public SocketWrench();
```

Example

The following example demonstrates creating an instance of the **SocketWrench** class object and resolving a hostname into an Internet address using the [Resolve](#) method.

```
Dim Socket As SocketTools.SocketWrench
Dim strHostName As String
Dim strHostAddress As String

Socket = New SocketTools.SocketWrench
strHostName = TextBox1.Text.Trim()

If Socket.Resolve(strHostName, strHostAddress) Then
    StatusBar1.Text = "The Internet address for " + strHostName + " is " +
strHostAddress
Else
    StatusBar1.Text = "The Internet address for " + strHostName + " could not be
resolved"
End If
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench Fields

The fields of the **SocketWrench** class are listed below. For a complete list of **SocketWrench** class members, see the [SocketWrench Members](#) topic.

Public Instance Fields

AdapterAddress	Returns the IP address associated with the specified network adapter.
HostAlias	Returns the aliases for a given host name.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.AdapterAddress Field

Returns the IP address associated with the specified network adapter.

[Visual Basic]

Public **ReadOnly** AdapterAddress As [AdapterAddressArray](#)

[C#]

public **readonly** [AdapterAddressArray](#) AdapterAddress;

Remarks

The **AdapterAddress** array returns the IP addresses that are associated with the local network or remote dial-up network adapters configured on the system. The **AdapterCount** property can be used to determine the number of adapters that are available.

Multihomed systems with more than one local network adapter, or a combination of local and dial-up adapters will not be listed in a specific order. An application should not make the assumption that the first address returned by **AdapterAddress** always refers to a local network adapter.

Note that it is possible that the **AdapterCount** property will return 0, and **AdapterAddress** will return an empty string. This indicates that the system does not have a physical network adapter with an assigned IP address, and there are no dial-up networking connections currently active. If a dial-up networking connection is established at some later point, the **AdapterCount** property will change to 1, and the **AdapterAddress** property will return the IP address allocated for that connection.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [AdapterAddressArray Class](#) | [AdapterCount Property](#)

SocketWrench.AdapterAddressArray Class

The class used to return the network adapters configured for the local host.

For a list of all members of this type, see [SocketWrench.AdapterAddressArray Members](#).

System.Object

SocketTools.SocketWrench.AdapterAddressArray

[Visual Basic]

```
<DefaultMember(MemberName:="Item")>  
Public Class SocketWrench.AdapterAddressArray
```

[C#]

```
[DefaultMember(MemberName="Item")]  
public class SocketWrench.AdapterAddressArray
```

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

The **AdapterAddressArray** class returns the network adapters configured for the local host. A read-only instance of this class is created by the SocketWrench class constructor and accessed using the **AdapterAddress** array.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketWrench.AdapterAddressArray Members](#) | [SocketTools Namespace](#) | [AdapterAddress Field](#)

SocketWrench.AdapterAddressArray Members

[SocketWrench.AdapterAddressArray overview](#)

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Returns the address for the specified network adapter.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[SocketWrench.AdapterAddressArray Class](#) | [SocketTools Namespace](#) | [AdapterAddress Field](#)

SocketWrench.AdapterAddressArray Properties

The properties of the **SocketWrench.AdapterAddressArray** class are listed below. For a complete list of **SocketWrench.AdapterAddressArray** class members, see the [SocketWrench.AdapterAddressArray Members](#) topic.

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Returns the address for the specified network adapter.

See Also

[SocketWrench.AdapterAddressArray Class](#) | [SocketTools Namespace](#) | [AdapterAddress Field](#)

SocketWrench.AdapterAddressArray.Count Property

Returns the maximum number of elements in the array.

[Visual Basic]

Public ReadOnly Property Count As Integer

[C#]

public int Count {get;}

Property Value

An integer value.

Remarks

This property will return the number of network adapters that are configured for the local system.

See Also

[SocketWrench.AdapterAddressArray Class](#) | [SocketTools Namespace](#)

SocketWrench.AdapterAddressArray.Item Property

Returns the address for the specified network adapter.

[Visual Basic]

```
Public Default ReadOnly Property Item( _  
    ByVal index As Integer _  
) As String
```

[C#]

```
public string this[  
    int index  
] {get;}
```

Parameters

index

An integer value which specifies the index into the array.

Property Value

A string which specifies an Internet address using dot notation.

See Also

[SocketWrench.AdapterAddressArray Class](#) | [SocketTools Namespace](#)

SocketWrench.HostAlias Field

Returns the aliases for a given host name.

[Visual Basic]

```
Public ReadOnly HostAlias As HostAliasArray
```

[C#]

```
public readonly HostAliasArray HostAlias;
```

Remarks

The **HostAlias** field returns an array of aliases assigned to the host specified by the **HostAddress** or **HostName** properties. If the host address or name can be resolved, the first element in the **HostAlias** array always refers to the host's fully qualified domain name.

Example

```
Dim nIndex As Integer

ListBox1.Items.Clear()
Socket.HostName = strHostName

For nIndex = 0 To Socket.HostAliases - 1
    ListBox1.Items.Add(Socket.HostAlias(nIndex))
Next
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [HostAliasArray Class](#) | [HostAliases](#) | [HostName](#)

SocketWrench.HostAliasArray Class

The class used to return the aliases for a specified host name.

For a list of all members of this type, see [SocketWrench.HostAliasArray Members](#).

System.Object

SocketTools.SocketWrench.HostAliasArray

[Visual Basic]

```
<DefaultMember(MemberName:="Item")>  
Public Class SocketWrench.HostAliasArray
```

[C#]

```
[DefaultMember(MemberName="Item")]  
public class SocketWrench.HostAliasArray
```

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

The **HostAliasArray** class returns the aliases for a specified host name. A read-only instance of this class is created by the SocketWrench class constructor and accessed using the **HostAlias** field.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketWrench.HostAliasArray Members](#) | [SocketTools Namespace](#) | [HostAlias Field](#) | [HostAliases](#) | [HostName](#)

SocketWrench.HostAliasArray Members

[SocketWrench.HostAliasArray overview](#)

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Returns an alias for the specified host name.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[SocketWrench.HostAliasArray Class](#) | [SocketTools Namespace](#) | [HostAlias Field](#) | [HostAliases](#) | [HostName](#)

SocketWrench.HostAliasArray Properties

The properties of the **SocketWrench.HostAliasArray** class are listed below. For a complete list of **SocketWrench.HostAliasArray** class members, see the [SocketWrench.HostAliasArray Members](#) topic.

Public Instance Properties

Count	Returns the maximum number of elements in the array.
Item	Returns an alias for the specified host name.

See Also

[SocketWrench.HostAliasArray Class](#) | [SocketTools Namespace](#) | [HostAlias Field](#) | [HostAliases](#) | [HostName](#)

SocketWrench.HostAliasArray.Count Property

Returns the maximum number of elements in the array.

[Visual Basic]

Public ReadOnly Property Count As Integer

[C#]

public int Count {get;}

Property Value

An integer value.

Remarks

This property will return the same value as the **HostAliases** property and is used to determine the maximum index value for the **HostAlias** array.

See Also

[SocketWrench.HostAliasArray Class](#) | [SocketTools Namespace](#)

SocketWrench.HostAliasArray.Item Property

Returns an alias for the specified host name.

[Visual Basic]

```
Public Default ReadOnly Property Item( _  
    ByVal index As Integer _  
) As String
```

[C#]

```
public string this[  
    int index  
] {get;}
```

Parameters

index

An integer value which specifies the index into the array.

Property Value

A string which specifies an alias for the host name.

See Also

[SocketWrench.HostAliasArray Class](#) | [SocketTools Namespace](#)

SocketWrench Properties

The properties of the **SocketWrench** class are listed below. For a complete list of **SocketWrench** class members, see the [SocketWrench Members](#) topic.

Public Instance Properties

AdapterCount	Get the number of available local and remote network adapters.
AddressFamily	Gets and sets a value that determines which version of the Internet Protocol will be used.
AtMark	Get a value that indicates if the next receive will return urgent data.
AutoResolve	Gets and sets a value that determines if host names and addresses are automatically resolved.
Backlog	Gets and sets a value that indicates the number of connections that may be queued for a listening socket.
Blocking	Gets and sets a value which indicates if the socket is in blocking mode.
Broadcast	Gets and sets a value which indicates if datagrams will be broadcast over the local network.
ByteOrder	Gets and sets a value which indicates how integer data is read and written to the socket.
CertificateExpires	Get a value that specifies the date that the security certificate expires.
CertificateIssued	Get a value that specifies the date that the security certificate was issued.
CertificateIssuer	Get a value that provides information about the organization that issued the certificate.
CertificateName	Gets and sets a value that specifies the name of the security certificate.
CertificatePassword	Gets and sets the password associated with the security certificate.
CertificateStatus	Gets a value which indicates the status of the security certificate returned by the remote host.
CertificateStore	Gets and sets a value that specifies the name of the local certificate store.
CertificateSubject	Gets a value that provides information about the organization that the server certificate was issued to.
CertificateUser	Gets and sets the user that owns the security certificate.
CipherStrength	Gets a value that indicates the length of the key used by the encryption algorithm for a secure

	connection.
CodePage	Gets and sets the code page used when reading and writing text.
ExternalAddress	Gets a value that specifies the external Internet address for the local system.
Handle	Gets a value that specifies the socket handle allocated for the current session.
HashStrength	Gets a value which specifies the length of the message digest that was selected for a secure connection.
HostAddress	Gets and sets a value which specifies the Internet address used to establish a connection.
HostAliases	Returns the number of aliases for the current host name.
HostFile	Gets and sets a value that specifies the name of a host file used to resolve host names and addresses.
HostName	Gets and sets a value which specifies the host name used to establish a connection.
InLine	Gets and sets a value that indicates if urgent data is received in-line with non-urgent data.
IsBlocked	Gets a value which indicates if the current thread is performing a blocking socket operation.
IsClosed	Gets a value which indicates if the connection to the remote host has been closed.
IsConnected	Gets a value which indicates if a connection to the remote host has been established.
IsInitialized	Gets a value which indicates if the current instance of the class has been initialized successfully.
IsListening	Gets a value which indicates if the socket is listening for client connections.
IsReadable	Gets a value which indicates if there is data available to be read from the socket.
IsWritable	Gets a value which indicates if data can be written to the socket without blocking.
KeepAlive	Gets and sets a value which indicates if keep-alive packets are sent on a connected socket.
LastError	Gets and sets a value which specifies the last error that has occurred.
LastErrorString	Gets a value which describes the last error that has occurred.
Linger	Gets and sets a value which specifies the number of seconds to wait for the socket to disconnect from the remote host.

LocalAddress	Gets and sets the local Internet address that the socket will be bound to.
LocalName	Gets a value which specifies the host name for the local system.
LocalPort	Gets and sets a value which specifies the local port number the socket will be bound to.
LocalService	Gets and sets a value which specifies the local service the socket will be bound to.
NoDelay	Gets and sets a value which specifies if the Nagle algorithm should be enabled or disabled.
Options	Gets and sets a value which specifies one or more socket options.
PeerAddress	Gets a value that specifies the Internet address of the remote host.
PeerName	Gets a value that specifies the name of the remote host.
PeerPort	Gets a value that specifies the port number used by the remote host.
PhysicalAddress	Gets a value which specifies the MAC address for the local system's network adapter.
Protocol	Gets and sets a value which specifies the socket protocol.
RemotePort	Gets and sets a value which specifies the remote port number.
RemoteService	Gets and sets a value which specifies the remote service.
ReservedPort	Gets and sets a value which indicates if a reserved port number was used.
ReuseAddress	Gets and sets a value which indicates if a socket address can be reused.
Route	Gets and sets a value which indicates if packets should be routed.
Secure	Gets and sets a value which specifies if a secure connection is established.
SecureCipher	Gets a value that specifies the encryption algorithm used for a secure connection.
SecureHash	Gets a value that specifies the message digest algorithm used for a secure connection.
SecureKeyExchange	Gets a value that specifies the key exchange algorithm used for a secure connection.
SecureProtocol	Gets and sets a value which specifies the protocol used for a secure connection.

Status	Gets a value which specifies the current status of the socket.
ThreadModel	Gets and sets a value which specifies the threading model for the class instance.
ThrowError	Gets and sets a value which specifies if method calls should throw exceptions when an error occurs.
Timeout	Gets and sets a value which specifies a timeout period in seconds.
Trace	Gets and sets a value which indicates if network function logging is enabled.
TraceFile	Gets and sets a value which specifies the name of the network function tracing logfile.
TraceFlags	Gets and sets a value which specifies the network function tracing flags.
Urgent	Gets and sets a value which specifies if urgent data will be read or written.
Version	Gets a value which returns the current version of the SocketWrench class library.
VpnAdapter	Gets the name of the active Virtual Private Network adapter.
VpnAddress	Gets the IP address of the active Virtual Private Network adapter.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.AdapterCount Property

Get the number of available local and remote network adapters.

[Visual Basic]

Public ReadOnly Property AdapterCount As Integer

[C#]

public int AdapterCount {get;}

Property Value

Returns the number of available local and remote network adapters.

Remarks

The **AdapterCount** property returns the number of local and remote dial-up networking adapters available on the local system. This value can be used in conjunction with the **AdapterAddress** array to enumerate the IP addresses assigned to the various network adapters.

Note that it is possible that the **AdapterCount** property will return 0, and **AdapterAddress** will return an empty string. This indicates that the system does not have a physical network adapter with an assigned IP address, and there are no dial-up networking connections currently active. If a dial-up networking connection is established at some later point, the **AdapterCount** property will change to 1, and the **AdapterAddress** property will return the IP address allocated for that connection.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [AdapterAddress Field](#)

SocketWrench.AtMark Property

Get a value that indicates if the next receive will return urgent data.

[Visual Basic]

```
Public ReadOnly Property AtMark As Boolean
```

[C#]

```
public bool AtMark {get;}
```

Property Value

Returns **true** if the next read on the socket will return urgent data.

Remarks

This property can only be used if the **Protocol** property is set to **SocketProtocol.socketStream** and the **InLine** property has been set to **true**.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.AutoResolve Property

Gets and sets a value that determines if host names and addresses are automatically resolved.

[Visual Basic]

Public Property AutoResolve As Boolean

[C#]

public bool AutoResolve {get; set;}

Property Value

Returns **true** if host names are automatically resolved to Internet addresses. The default value is **false**.

Remarks

Setting the **AutoResolve** property determines if the class automatically resolves host names and addresses specified by the **HostName** and **HostAddress** properties. If set to **true**, setting the **HostName** property will cause the class to automatically determine the corresponding IP address and update the **HostAddress** property accordingly. Likewise, setting the **HostAddress** property will cause the class to determine the host name and update the **HostName** property. Setting this property to **false** prevents the class from resolving host names until a connection attempt is made.

It is important to note that setting the **HostName** or **HostAddress** property may cause the current thread to block, sometimes for several seconds, until the name or address is resolved. To prevent this behavior, set this property value to **false**.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Backlog Property

Gets and sets a value that indicates the number of connections that may be queued for a listening socket.

[Visual Basic]

Public Property Backlog As Integer

[C#]

public int Backlog {get; set;}

Property Value

Returns an integer value that specifies the size of the backlog queue. The default value is 5.

Remarks

The **Backlog** property specifies the maximum size of the queue used to manage pending connections to the service. If the property is set to value which exceeds the maximum size for the underlying service provider, it will be silently adjusted to the nearest legal value. There is no standard way to determine what the maximum backlog value is.

This property must be set to the desired value before the **Listen** method is called, if the **Listen** method is used with default parameters.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Blocking Property

Gets and sets a value which indicates if the socket is in blocking mode.

[Visual Basic]

Public Property Blocking As Boolean

[C#]

public bool Blocking {get; set;}

Property Value

Returns **true** if the socket is in blocking mode; otherwise it returns **false**. The default value is **true**.

Remarks

Setting the **Blocking** property determines if socket operations complete synchronously or asynchronously. If set to **true**, then each socket operation (such as sending or receiving data) will return when the operation has completed or timed-out. If set to **false**, socket operations will return immediately. If the operation would result in the socket blocking (such as attempting to read data when no data has been sent by the remote host), an error is generated.

It is important to note that certain events, such as **OnDisconnect**, **OnRead** and **OnWrite** are only fired if the socket is in non-blocking mode.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Broadcast Property

Gets and sets a value which indicates if datagrams will be broadcast over the local network.

[Visual Basic]

Public Property Broadcast As Boolean

[C#]

public bool Broadcast {get; set;}

Property Value

Returns **true** if datagrams will be broadcast; otherwise returns **false**. The default value is **false**.

Remarks

If the **Broadcast** property is set to a value of **true**, the datagram written to the socket will be broadcast to all systems on the network. Use of this property is restricted to the UDP protocol and the value is ignored for TCP connections.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.ByteOrder Property

Gets and sets a value which indicates how integer data is read and written to the socket.

[Visual Basic]

Public Property ByteOrder As [SocketByteOrder](#)

[C#]

public [SocketWrench.SocketByteOrder](#) ByteOrder {get; set;}

Property Value

A [SocketByteOrder](#) enumeration value which specifies the byte order. The default is **Native**.

Remarks

The **ByteOrder** property is used to specify how integer data is written to and read from the socket. The default value for this property is **Native**, which specifies that integers should be written in the native byte order for the local machine. A value of **Network** indicates that integers should be written in network byte order.

When applications write integer values on a socket (instead of string representations of those values), they should typically be converted to network byte order before they are sent. Likewise, when an integer value is read, it should then be converted from the network byte order back to the byte order used by the local machine. The native byte order, also called the host byte order, should only be used if it can be assured that both the sender and the receiver are running on an identical or compatible machine architectures (for example, if both systems are Intel-based).

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.CertificateExpires Property

Get a value that specifies the date that the security certificate expires.

[Visual Basic]

```
Public ReadOnly Property CertificateExpires As String
```

[C#]

```
public string CertificateExpires {get;}
```

Property Value

A string which specifies a date using the local date and time format.

Remarks

The **CertificateExpires** property returns a string that specifies the date and time that the security certificate expires. This property will return an empty string if a secure connection has not been established with the remote host.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.CertificateIssued Property

Get a value that specifies the date that the security certificate was issued.

[Visual Basic]

Public ReadOnly Property CertificateIssued As String

[C#]

public string CertificateIssued {get;}

Property Value

A string which specifies a date using the local date and time format.

Remarks

The **CertificateIssued** property returns a string that specifies the date and time that the security certificate was issued. This property will return an empty string if a secure connection has not been established with the remote host.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.CertificateIssuer Property

Get a value that provides information about the organization that issued the certificate.

```
[Visual Basic]
Public ReadOnly Property CertificateIssuer As String
```

```
[C#]
public string CertificateIssuer {get;}
```

Property Value

A string that contains a comma separated list of name value pairs.

Remarks

The **CertificateIssuer** property returns a string that contains information about the organization that issued the server certificate. The string value is a comma separated list of tagged name and value pairs. In the nomenclature of the X.500 standard, each of these pairs are called a relative distinguished name (RDN), and when concatenated together, forms the issuer's distinguished name (DN). For example:

```
C=US, O="RSA Data Security, Inc.", OU=Secure Server Certification Authority
```

To obtain a specific value, such as the name of the issuer or the issuer's country, the application must parse the string returned by this property. Some of the common tokens used in the distinguished name are:

Token	Description
C	The ISO standard two character country code.
S	The name of the state or province.
L	The name of the city or locality.
O	The name of the company or organization.
OU	The name of the department or organizational unit
CN	The common name; with X.509 certificates, this is the domain name of the site the certificate was issued for.

This property will return an empty string if a secure connection has not been established with the remote host.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.CertificateName Property

Gets and sets a value that specifies the name of the security certificate.

[Visual Basic]

Public Property CertificateName As String

[C#]

public string CertificateName {get; set;}

Property Value

A string which specifies the certificate name.

Remarks

The **CertificateName** property sets the common name or friendly name of the certificate that should be used when establishing a secure client connection or accepting a secure connection from a remote host. This property is used in conjunction with the **CertificateStore** property to identify the client or server certificate.

For client applications, it is only required that you set this property value if the server requires a client certificate for authentication. If this property is not set, a client certificate will not be provided to the server. The certificate must be designated as a client certificate and have a private key associated with it, otherwise the connection attempt will fail.

For server applications, it is required that you specify a certificate name if security has been enabled by setting the **Secure** property to true. The certificate must be designated as a server certificate and have a private key associated with it, otherwise incoming client connections cannot be accepted.

When the certificate store is searched for a matching certificate, it will first search for any certificate with a friendly name that matches the property value. If no valid certificate is found, it will then search for a certificate with a matching common name.

Certificates may be installed and viewed on the local system using the Certificate Manager that is included with the Windows operating system. For more information, refer to the documentation for the Microsoft Management Console.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [CertificateStore Property](#) | [Secure Property](#)

SocketWrench.CertificateStatus Property

Gets a value which indicates the status of the security certificate returned by the remote host.

[Visual Basic]

Public ReadOnly Property CertificateStatus As [SecurityCertificate](#)

[C#]

public [SocketWrench.SecurityCertificate](#) CertificateStatus {get;}

Property Value

A [SecurityCertificate](#) enumeration value which specifies the status of the certificate.

Remarks

The **CertificateStatus** property is used to determine the status of the security certificate returned by the remote host when a secure connection has been established. This property value should be checked after the connection to the server has completed, but prior to beginning a transaction.

Note that if the certificate cannot be validated, the secure connection will not be automatically terminated. It is the responsibility of your application to determine the best course of action to take if the certificate is invalid. Even if the security certificate cannot be validated, the data exchanged with the remote host will still be encrypted.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.CertificateStore Property

Gets and sets a value that specifies the name of the local certificate store.

[Visual Basic]

Public Property CertificateStore As String

[C#]

public string CertificateStore {get; set;}

Property Value

A string which specifies the certificate store name. The default value is the current user's personal certificate store.

Remarks

The **CertificateStore** property is used to specify the name of the certificate store which contains the security certificate to use when establishing a secure connection. The certificate may either be stored in the registry or in a file. If the certificate is stored in the registry, then this property should be set to one of the following predefined values:

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. If a certificate store is not specified, this is the default value that is used.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

In most cases the client certificate will be installed in the user's personal certificate store, and therefore it is not necessary to set this property value because that is the default location that will be used to search for the certificate. This property is only used if the **CertificateName** property is also set to a valid certificate name.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU" for the current user, or "HKLM" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, it will default to the certificate store for the current user.

This property may also be used to specify a file that contains the client certificate. In this case, the property should specify the full path to the file and must contain both the certificate and private key in PKCS #12 format. If the file is protected by a password, the **CertificatePassword** property must also be set to specify the password.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [CertificatePassword Property](#) | [Secure Property](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketWrench.CertificateSubject Property

Gets a value that provides information about the organization that the server certificate was issued to.

[Visual Basic]

Public ReadOnly Property CertificateSubject As String

[C#]

public string CertificateSubject {get;}

Property Value

A string that contains a comma separated list of name value pairs.

Remarks

The **CertificateSubject** property returns a string that contains information about the organization that the server certificate was issued to. The string value is a comma separated list of tagged name and value pairs. In the nomenclature of the X.500 standard, each of these pairs are called a relative distinguished name (RDN), and when concatenated together, forms the issuer's distinguished name (DN). For example:

C=US, O="RSA Data Security, Inc.", OU=Secure Server Certification Authority

To obtain a specific value, such as the name of the issuer or the issuer's country, the application must parse the string returned by this property. Some of the common tokens used in the distinguished name are:

Token	Description
C	The ISO standard two character country code.
S	The name of the state or province.
L	The name of the city or locality.
O	The name of the company or organization.
OU	The name of the department or organizational unit
CN	The common name; with X.509 certificates, this is the domain name of the site the certificate was issued for.

This property will return an empty string if a secure connection has not been established with the remote host.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.CipherStrength Property

Gets a value that indicates the length of the key used by the encryption algorithm for a secure connection.

[Visual Basic]

Public ReadOnly Property CipherStrength As Integer

[C#]

public int CipherStrength {get;}

Property Value

An integer value which specifies the encryption key length if a secure connection has been established; otherwise a value of 0 is returned.

Remarks

The **CipherStrength** property returns the number of bits in the key used to encrypt the secure data stream. Common values returned by this property are 128 and 256. A key length of 40 or 56 bits is considered insecure and subject to brute force attacks. 128-bit and 256-bit keys are considered secure. If this property returns a value of 0, this means that a secure connection has not been established with the remote host.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.CodePage Property

Gets and sets the code page used when reading and writing text.

[Visual Basic]

Public Property CodePage As Integer

[C#]

public int CodePage {get; set;}

Property Value

An integer value which specifies the current code page. A value of zero specifies the default code page for the current locale should be used. To preserve the original Unicode text, you can use code page 65001 which specifies UTF-8 character encoding.

Remarks

All data which is exchanged over a socket is sent and received as 8-bit bytes, typically referred to as "octets" in networking terminology. However, strings in .NET are Unicode where each character is represented by 16 bits. To send and receive data using strings, these Unicode strings are converted to a stream of bytes.

By default, strings are converted to an array of bytes using the code page for the current locale, mapping the 16-bit Unicode characters to bytes. Similarly, when reading data from the socket into a string buffer, the stream of bytes received from the remote host are converted to Unicode before they are returned to your application.

If you are exchanging text with another system and it appears to be corrupted or characters are being replaced with question marks or other symbols, it is likely the system is sending text which is using a different character encoding. Most services use UTF-8 encoding to represent non-ASCII characters and selecting the UTF-8 code page will typically resolve the issue.

Strings are only guaranteed to be safe when sending and receiving text. Using a string data type is not recommended when reading or writing binary data to a socket. If possible, you should always use a byte array as the buffer parameter for the Read and Write methods whenever you are exchanging binary data.

For backwards compatibility, this class defaults to using the code page for the current locale. This property value directly corresponds to Windows code page identifiers, and will accept any valid code page supported by the .NET Framework. Setting this property to an invalid code page will generate an exception.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.ExternalAddress Property

Gets a value that specifies the external Internet address for the local system.

[Visual Basic]

```
Public ReadOnly Property ExternalAddress As String
```

[C#]

```
public string ExternalAddress {get;}
```

Property Value

A string which specifies an Internet address using dotted notation.

Remarks

The **ExternalAddress** property returns the IP address assigned to the router that connects the local host to the Internet. This is typically used by an application executing on a system in a local network that uses a router which performs Network Address Translation (NAT). In that network configuration, the **LocalAddress** property will only return the IP address for the local system on the LAN side of the network unless a connection has already been established to a remote host. The **ExternalAddress** property can be used to determine the IP address assigned to the router on the Internet side of the connection and can be particularly useful for servers running on a system behind a NAT router.

Using this property requires that you have an active connection to the Internet; checking the value of this property on a system that uses dial-up networking may cause the operating system to automatically connect to the Internet service provider. The class may be unable to determine the external IP address for the local host for a number of reasons, particularly if the system is behind a firewall or uses a proxy server that restricts access to external sites on the Internet. If the external address for the local host cannot be determined, the property will return an empty string.

If the class is able to obtain a valid external address for the local host, that address will be cached for sixty minutes. Because dial-up connections typically have different IP addresses assigned to them each time the system is connected to the Internet, it is recommended that this property only be used in conjunction with broadband connections using a NAT router.

It is important to note that checking this property value may cause the current thread to block until the external IP address can be resolved and should never be used in conjunction with non-blocking (asynchronous) socket connections. If you need to check this property value in an application which uses asynchronous sockets, it is recommended that you create a new thread and access the property from within that thread.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Handle Property

Gets a value that specifies the socket handle allocated for the current session.

[Visual Basic]

```
Public ReadOnly Property Handle As Integer
```

[C#]

```
public int Handle {get;}
```

Property Value

An integer which represents a socket handle. If there is no active connection, a value of -1 is returned.

Remarks

The **Handle** property specifies the socket descriptor of the listening socket. To accept the connection, a new instance of the SocketWrench class should be created, passing this value to the **Accept** method in the new class instance.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.HashStrength Property

Gets a value which specifies the length of the message digest that was selected for a secure connection.

[Visual Basic]

```
Public ReadOnly Property HashStrength As Integer
```

[C#]

```
public int HashStrength {get;}
```

Property Value

An integer value which specifies the length of the message digest if a secure connection has been established; otherwise a value of 0 is returned.

Remarks

The **HashStrength** property returns the number of bits used in the message digest (hash) that was selected. Common values returned by this property are 128 and 160. If this property returns a value of 0, this means that a secure connection has not been established with the remote host.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.HostAddress Property

Gets and sets a value which specifies the Internet address used to establish a connection.

[Visual Basic]

```
Public Property HostAddress As String
```

[C#]

```
public string HostAddress {get; set;}
```

Property Value

A string which specifies an Internet address using dotted notation.

Remarks

The **HostAddress** property can be used to set the Internet address for a remote system that you wish to communicate with. If the **AutoResolve** property is set to **true** and the address is assigned to a valid host name, the **HostName** property will be updated with that value.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.HostFile Property

Gets and sets a value that specifies the name of a host file used to resolve host names and addresses.

[Visual Basic]

```
Public Property HostFile As String
```

[C#]

```
public string HostFile {get; set;}
```

Property Value

A string which specifies a file name.

Remarks

The **HostFile** property is used to specify the name of an alternate file for resolving hostnames and IP addresses. The host file is used as a database that maps an IP address to one or more hostnames, and is used when setting the **HostName** or **HostAddress** properties and establishing a connection with a remote host. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Setting this property loads the file into memory allocated for the current thread. If the contents of the file have changed after the function has been called, those changes will not be reflected when resolving hostnames or addresses. To reload the host file from disk, set the property again with the same file name. To remove the alternate host file from memory, specify an empty string as the file name.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

Because the alternate host file is cached for the current thread, setting this property will affect all instances of the class in the same thread. For example, if a project has created three instances of the class, setting the HostFile property will affect all three instances, not just the instance that set the property. To determine if an alternate host file has been cached, check the property value. If the property returns an empty string, no alternate host file has been cached.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.HostName Property

Gets and sets a value which specifies the host name used to establish a connection.

[Visual Basic]

```
Public Property HostName As String
```

[C#]

```
public string HostName {get; set;}
```

Property Value

A string which specifies a host name.

Remarks

The **HostName** property can be used to set the host name for a remote system that you wish to communicate with. If the **AutoResolve** property is set to **true** and the name can be resolved to a valid Internet address, the **HostAddress** property will be updated with that value.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.InLine Property

Gets and sets a value that indicates if urgent data is received in-line with non-urgent data.

[Visual Basic]

```
Public Property InLine As Boolean
```

[C#]

```
public bool InLine {get; set;}
```

Property Value

Returns **true** if urgent data will be received in-line; otherwise returns **false**. The default value is **false**.

Remarks

The **InLine** property controls how urgent (out-of-band) data is handled when reading data from the socket. If set to a value of **true**, urgent data is placed in the data stream along with non-urgent data. To determine if the data that is being read is urgent, the **AtMark** property can be read.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.IsBlocked Property

Gets a value which indicates if the current thread is performing a blocking socket operation.

[Visual Basic]

```
Public ReadOnly Property IsBlocked As Boolean
```

[C#]

```
public bool IsBlocked {get;}
```

Property Value

Returns **true** if the current thread is blocking, otherwise returns **false**.

Remarks

The **IsBlocked** property returns **true** if the current thread is blocked performing an operation. Because the Windows Sockets API only permits one blocking operation per thread of execution, this property should be checked before starting any blocking operation in response to an event.

If the **IsBlocked** property returns **false**, this means there are no blocking operations on the current thread at that time. However, this does not guarantee that the next socket operation will not fail. An application should always check the return value from a socket operation and check the value of the **LastError** property if an error occurs.

Note that this property will return **true** if there is any blocking operation being performed by the current thread, regardless of whether this specific instance of the class is responsible for the blocking operation or not.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.IsClosed Property

Gets a value which indicates if the connection to the remote host has been closed.

[Visual Basic]

Public ReadOnly Property IsClosed As Boolean

[C#]

public bool IsClosed {get;}

Property Value

Returns **true** if the connection has been closed; otherwise returns **false**.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.IsConnected Property

Gets a value which indicates if a connection to the remote host has been established.

[Visual Basic]

Public ReadOnly Property IsConnected As Boolean

[C#]

public bool IsConnected {get;}

Property Value

Returns **true** if the connection has been established; otherwise returns **false**.

Remarks

The **IsConnected** property can only be used to indicate if there is still a logical connection to the remote host. It cannot be used to detect abnormal conditions such as the remote host aborting the connection, the physical network connection being lost or other critical errors.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.IsInitialized Property

Gets a value which indicates if the current instance of the class has been initialized successfully.

[Visual Basic]

Public ReadOnly Property IsInitialized As Boolean

[C#]

public bool IsInitialized {get;}

Property Value

Returns **true** if the class instance has been initialized; otherwise returns **false**.

Remarks

The **IsInitialized** property is used to determine if the current instance of the class has been initialized properly. Normally this is done automatically by the class constructor, however there are circumstances where the class may not be able to initialize itself.

The most common reasons that a class instance may not initialize correctly is that no runtime license key has been defined in the assembly or the license key provided is invalid. It may also indicate a problem with the system configuration or user access rights, such as not being able to load the required networking libraries or not being able to access the system registry.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.IsListening Property

Gets a value which indicates if the socket is listening for client connections.

[Visual Basic]

Public ReadOnly Property IsListening As Boolean

[C#]

public bool IsListening {get;}

Property Value

Returns **true** if the socket is listening for client connections; otherwise returns **false**.

Remarks

The **IsListening** property will return **true** if the socket was created using the **Listen** method and it is currently accepting incoming client connections. In all other situations, this property will return **false**.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.IsReadable Property

Gets a value which indicates if there is data available to be read from the socket.

[Visual Basic]

```
Public ReadOnly Property IsReadable As Boolean
```

[C#]

```
public bool IsReadable {get;}
```

Property Value

Returns **true** if there is data available to be read; otherwise returns **false**.

Remarks

The **IsReadable** property returns **true** if data can be read from the socket without blocking. For non-blocking sockets, this property can be checked before the application attempts to read the socket. Note that even if this property does return **true** indicating that there is data available to be read, applications should always check the return value from the **Read** method.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.IsWritable Property

Gets a value which indicates if data can be written to the socket without blocking.

[Visual Basic]

```
Public ReadOnly Property IsWritable As Boolean
```

[C#]

```
public bool IsWritable {get;}
```

Property Value

Returns **true** if data can be written to the socket; otherwise returns **false**.

Remarks

The **IsWritable** property returns **true** if data can be written to the socket without blocking. For non-blocking sockets, this property can be checked before the application attempts to write data to the socket. Note that even if this property does return **true** indicating that data can be written to the socket, applications should always check the return value from the **Write** method.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.KeepAlive Property

Gets and sets a value which indicates if keep-alive packets are sent on a connected socket.

[Visual Basic]

Public Property KeepAlive As Boolean

[C#]

public bool KeepAlive {get; set;}

Property Value

Returns **true** if keep-alive packets are sent when the connection is idle, otherwise returns **false**. The default value is **false**.

Remarks

Setting the **KeepAlive** property to a value of **true** specifies that special packets are to be sent to the remote system when no data is being exchanged to ensure the connection remains active. This property can only be set for sockets that were created with the **Protocol** property set to a value of **SocketProtocol.protocolStream**.

If this property is set to **true**, keep-alive packets will start being generated five seconds after the socket has become idle with no data being sent or received. Enabling this option can be used by applications to detect when a physical network connection has been lost. However, it is recommended that most applications query the remote host directly to determine if the connection is still active. This is typically accomplished by sending specific commands to the server to query its status, or checking the elapsed time since the last response from the server.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.LastError Property

Gets and sets a value which specifies the last error that has occurred.

[Visual Basic]

Public Property LastError As [ErrorCode](#)

[C#]

public [SocketWrench.ErrorCode](#) LastError {get; set;}

Property Value

Returns an [ErrorCode](#) enumeration value which specifies the last error code.

Remarks

The **LastError** property returns the error code associated with the last error that occurred for the current instance of the class. It is important to note that this value only has meaning if the previous method indicates that an error has actually occurred.

It is possible to explicitly clear the last error code by assigning the property to the value **ErrorCode.None**.

The error code value can be cast to an integer value for display purposes if required. For a description of the error that can be displayed using a message box or some other similar mechanism, get the value of the **LastErrorString** property.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.LastErrorString Property

Gets a value which describes the last error that has occurred.

[Visual Basic]

```
Public ReadOnly Property LastErrorString As String
```

[C#]

```
public string LastErrorString {get;}
```

Property Value

A string which describes the last error that has occurred.

Remarks

The **LastErrorString** property can be used to obtain a description of the last error that occurred for the current instance of the class. It is important to note that this value only has meaning if the previous method indicates that an error has actually occurred.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Linger Property

Gets and sets a value which specifies the number of seconds to wait for the socket to disconnect from the remote host.

[Visual Basic]

Public Property Linger As Integer

[C#]

public int Linger {get; set;}

Property Value

An integer value which specifies a number of seconds. The default value is 0.

Remarks

Setting the **Linger** property to a value greater than zero indicates that the **Disconnect** method should wait up to the specified number of seconds for any data on the socket to be written before it is closed. A value of zero indicates that the socket should be closed immediately (but gracefully, without data loss).

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.LocalAddress Property

Gets and sets the local Internet address that the socket will be bound to.

[Visual Basic]

Public Property LocalAddress As String

[C#]

public string LocalAddress {get; set;}

Property Value

A string which specifies an Internet address in dotted notation.

Remarks

The **LocalAddress** property is used to specify the local Internet address that the socket will be bound to when a connection is established with a remote host. By default this property is not assigned a value, which specifies that the socket should be bound to any appropriate network interface on the local system.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.LocalName Property

Gets a value which specifies the host name for the local system.

[Visual Basic]

```
Public ReadOnly Property LocalName As String
```

[C#]

```
public string LocalName {get;}
```

Property Value

A string which specifies the local host name.

Remarks

The **LocalName** property returns the fully-qualified host name assigned to the local system. If the system has not been configured with an Internet domain name, then this property will return the NetBIOS name assigned to the local system.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.LocalPort Property

Gets and sets a value which specifies the local port number the socket will be bound to.

[Visual Basic]

```
Public Property LocalPort As Integer
```

[C#]

```
public int LocalPort {get; set;}
```

Property Value

An integer value which specifies a port number. The default value is 0.

Remarks

The **LocalPort** property is used to specify the local port number that the socket will be bound to when a connection is established with a remote host. By default this property value is 0, which specifies that the socket should be bound to any appropriate port number that is available on the local system. After a connection has been established, this property will return the actual port number that was allocated for the socket.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.NoDelay Property

Gets and sets a value which specifies if the Nagle algorithm should be enabled or disabled.

[Visual Basic]

```
Public Property NoDelay As Boolean
```

[C#]

```
public bool NoDelay {get; set;}
```

Property Value

Returns **true** if the Nagle algorithm has been disabled; otherwise it returns **false**. The default value is **false**.

Remarks

The **NoDelay** property is used to enable or disable the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host. By default this property value is set to **false**, which enables the Nagle algorithm (in other words, the data being written may not actually be sent until it is optimal to do so). Setting this property to **true** disables the Nagle algorithm, maintaining the time delays between the data packets being sent.

This property should be set to **true** only if it is absolutely required and the implications of doing so are understood. Disabling the Nagle algorithm can have a significant negative impact on the performance of your application.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Options Property

Gets and sets a value which specifies one or more socket options.

[Visual Basic]

Public Property Options As [SocketOptions](#)

[C#]

public [SocketWrench.SocketOptions](#) Options {get; set;}

Property Value

Returns one or more [SocketOptions](#) enumeration flags which specify the options for the socket. The default value for this property is **None**.

Remarks

The **Options** property specifies one or more default socket options which are used when creating a socket using either the **Accept** or **Connect** methods.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.PeerAddress Property

Gets a value that specifies the Internet address of the remote host.

[Visual Basic]

```
Public ReadOnly Property PeerAddress As String
```

[C#]

```
public string PeerAddress {get;}
```

Property Value

A string which specifies an Internet address in dotted notation.

Remarks

The **PeerAddress** property returns the Internet address of the remote system that the local host is connected to. If a datagram socket is being used, this property will return the address of the system which sent the last datagram that was read. If no connection has been established, this property will return an empty string.

If this property is accessed inside an **OnAccept** event handler, it will return the address of the client that is requesting the connection. The application may use this information to determine if it wishes to accept or reject the client connection. If the address is not available to the client at that time, this property will return the address 0.0.0.0.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.PeerName Property

Gets a value that specifies the name of the remote host.

[Visual Basic]

```
Public ReadOnly Property PeerName As String
```

[C#]

```
public string PeerName {get;}
```

Property Value

A string which specifies the peer host name.

Remarks

The **PeerName** property returns the name of the remote system that the local host is connected to. If a datagram socket is being used, this property will return the name of the system which sent the last datagram that was read.

Accessing this property may cause the thread to block until the peer address can be resolved to a host name.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.PeerPort Property

Gets a value that specifies the port number used by the remote host.

[Visual Basic]

```
Public ReadOnly Property PeerPort As Integer
```

[C#]

```
public int PeerPort {get;}
```

Property Value

An integer value which specifies the peer port number.

Remarks

The **PeerName** property returns the port number of the remote system that the local host is connected to. If a datagram socket is being used, this property will return the port number of the remote host which sent the last datagram that was read.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.PhysicalAddress Property

Gets a value which specifies the MAC address for the local system's network adapter.

[Visual Basic]

```
Public ReadOnly Property PhysicalAddress As String
```

[C#]

```
public string PhysicalAddress {get;}
```

Property Value

A string which specifies the network adapter MAC address.

Remarks

The **PhysicalAddress** property returns the Media Access Control (MAC) address for an Ethernet or Token Ring network adapter installed and configured on the local system. Since it is guaranteed that every adapter is assigned a unique address throughout the world, this value can be safely used for identification purposes. It is possible that this property will return an empty string, which indicates that it could not find a network adapter.

If more than one physical network adapter is installed on the system, this property will return the MAC address of the first adapter that it finds.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Protocol Property

Gets and sets a value which specifies the socket protocol.

[Visual Basic]

Public Property Protocol As [SocketProtocol](#)

[C#]

public [SocketWrench.SocketProtocol](#) **Protocol** {get; set;}

Property Value

Returns a [SocketProtocol](#) enumeration value which specifies the socket protocol. The default value is **Stream**.

Remarks

The **Protocol** property specifies the type of socket that will be created. This property may only be set before the **Connect** method is called; attempting to change this property value after a connection has been established will generate an error.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.RemotePort Property

Gets and sets a value which specifies the remote port number.

[Visual Basic]

```
Public Property RemotePort As Integer
```

[C#]

```
public int RemotePort {get; set;}
```

Property Value

An integer value which specifies a port number.

Remarks

The **RemotePort** property is used to set the port number that will be used to establish a connection with a remote host. If the port number specifies a well-known port, the **RemoteService** property will be updated with that name.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.ReservedPort Property

Gets and sets a value which indicates if a reserved port number was used.

[Visual Basic]

```
Public Property ReservedPort As Boolean
```

[C#]

```
public bool ReservedPort {get; set;}
```

Property Value

Returns **true** if a reserved port number was used; otherwise returns **false**. The default value is **false**.

Remarks

The **ReservedPort** property determines if a reserved local port number is use when the socket is created (reserved port numbers are in the range of 513 through 1023, inclusive). Some application protocols require that the client bind to a local port number in this range. By setting the **LocalPort** property to 0 and the **ReservedPort** property to **true**, a reserved port number will be used when the socket is created. The default value for this property is **false**, which specifies that a standard port number with a value of 1024 or higher will be bound to the socket unless the **LocalPort** property is explicitly set to a non-zero value. Reserved ports should only be used by those applications that need them to implement a specific protocol.

It is possible that the error **AddressInUse** will be returned when attempting to connect using a reserved port number. The value of the **LocalPort** property will specify the reserved port number that could not be used.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.ReuseAddress Property

Gets and sets a value which indicates if a socket address can be reused.

[Visual Basic]

Public Property ReuseAddress As Boolean

[C#]

public bool ReuseAddress {get; set;}

Property Value

Returns **true** if an address can be reused; otherwise returns **false**. The default value is **true**.

Remarks

The **ReuseAddress** property determines if a socket can be bound to an address and port number that were recently used. If this property is **true**, then addresses can be reused as needed. If the property is **false**, then addresses cannot be reused and an error will be generated if the address was recently used by another socket.

This property is typically used by server applications. By setting the property to **true**, a server can be stopped and immediately restarted using the same port number; otherwise, the server must wait approximately two minutes before the port can be reused.

If you wish to determine if a local port number is already in use by another application, set this property to **false** and attempt to create a socket using that port number. If another application is already using that port number, an error will be generated indicating that the address is in use and the socket could not be created.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Route Property

Gets and sets a value which indicates if packets should be routed.

[Visual Basic]

Public Property Route As Boolean

[C#]

public bool Route {get; set;}

Property Value

Returns **true** if packets should be routed; otherwise returns **false**. The default value is **true**.

Remarks

The **Route** property determines if routing tables should be used when sending data. If the property is set to **false**, then packets will be sent directly to the network interface; if there is a router between the local and remote hosts, the data will be lost. It is not recommended that you change this property value unless it is required by your application and you fully understand the implications of doing so.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Secure Property

Gets and sets a value which specifies if a secure connection is established.

[Visual Basic]

```
Public Property Secure As Boolean
```

[C#]

```
public bool Secure {get; set;}
```

Property Value

Returns **true** if a secure connection is established; otherwise returns **false**. The default value is **false**.

Remarks

The **Secure** property determines if a secure connection is established with the remote host. The default value for this property is **false**, which specifies that a standard connection to the server is used. To establish a secure connection, the application should set this property value to **true** prior to calling the **Accept** or **Connect** methods. Once the connection has been established, the client may exchange data with the server as with standard connections.

It is possible for an application to establish a non-secure connection, and then switch to a secure connection at some later point during the session. Initially set the **Secure** property to **false**, then connect to the server normally. Once the connection has been established, setting the **Secure** property to **true** will cause the application to negotiate a secure connection with the remote host. If the socket was created using the **Accept** method, the class will block and wait for the client to begin the negotiation. If the socket was created using the **Connect** method, it will immediately begin the negotiation with the server. Note that if a non-blocking (asynchronous) socket is being used, the application must wait to set the **Secure** property to **true** after the **OnConnect** event has fired.

Setting the **Secure** property to **false** during a connection will cause the class to send a shutdown message to the remote host. This may cause the remote host to terminate the connection, however it will not close the socket. It is recommended that applications do not set the **Secure** property to **false** after a secure connection has been established, and instead use the **Disconnect** method to close the connection.

It is strongly recommended that any application that sets this property **true** use error handling to trap an errors that may occur. If the class is unable to initialize the security libraries, or otherwise cannot create a secure session for the client, an error will be generated when this property value is set.

Example

```
Socket.ThrowError = True
```

```
Try
```

```
    Socket.Secure = True
```

```
    Socket.Connect(strHostName, nHostPort, defTimeout)
```

```
    Socket.WriteLine("GET " + strFileName + " HTTP/1.0")
```

```
    Socket.WriteLine("Host: " + strHostName)
```

```
    Socket.WriteLine("Accept: text/*")
```

```
    Socket.WriteLine()
```

```
Do
```

```
    Socket.ReadLine(strBuffer)
```

```
Loop Until strBuffer.Length = 0
```

```
Socket.ReadStream(strBuffer, True)
```

```
Catch ex As SocketTools.SocketWrenchException
    MsgBox(ex.Message)
End Try

Socket.Disconnect()
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.SecureCipher Property

Gets a value that specifies the encryption algorithm used for a secure connection.

[Visual Basic]

```
Public ReadOnly Property SecureCipher As SecureCipherAlgorithm
```

[C#]

```
public SocketWrench.SecureCipherAlgorithm SecureCipher {get;}
```

Property Value

A [SecureCipherAlgorithm](#) enumeration value which identifies the algorithm.

Remarks

The **SecureCipher** property returns a value which identifies the algorithm used to encrypt the data stream. If a secure connection has not been established, this property will return a value of **None**.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.SecureHash Property

Gets a value that specifies the message digest algorithm used for a secure connection.

[Visual Basic]

Public **ReadOnly** **Property** SecureHash As [SecureHashAlgorithm](#)

[C#]

public [SocketWrench.SecureHashAlgorithm](#) SecureHash {get;}

Property Value

A [SecureHashAlgorithm](#) enumeration value which identifies the algorithm.

Remarks

The **SecureHash** property returns a value which identifies the message digest (hash) algorithm that was selected when a secure connection was established. If a secure connection has not been established, this property will return a value of **None**.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.SecureKeyExchange Property

Gets a value that specifies the key exchange algorithm used for a secure connection.

[Visual Basic]

Public ReadOnly Property SecureKeyExchange As [SecureKeyAlgorithm](#)

[C#]

public [SocketWrench.SecureKeyAlgorithm](#) SecureKeyExchange {get;}

Property Value

A [SecureKeyAlgorithm](#) enumeration value which identifies the algorithm.

Remarks

The **SecureKeyExchange** property returns a value which identifies the key exchange algorithm that was selected when a secure connection was established. If a secure connection has not been established, this property will return a value of **ExchangeNone**.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.SecureProtocol Property

Gets and sets a value which specifies the protocol used for a secure connection.

[Visual Basic]

Public Property SecureProtocol As [SecurityProtocols](#)

[C#]

public [SocketWrench.SecurityProtocols](#) SecureProtocol {get; set;}

Property Value

A [SecurityProtocols](#) enumeration value which identifies the protocol to be used when establishing a secure connection.

Remarks

The **SecureProtocol** property can be used to specify the security protocol to be used when establishing a secure connection with a server or accepting a secure connection from a client. By default, the class will attempt to use either TLS 1.2 or TLS 1.3 to establish the connection, with the appropriate protocol automatically selected based on the capabilities of the remote host. It is recommended that you only change this property value if you fully understand the implications of doing so. Assigning a value to this property will override the default protocol and force the class to attempt to use only the protocol specified.

Multiple security protocols may be specified by combining them using a bitwise **or** operator. After a connection has been established, this property will identify the protocol that was selected. Attempting to set this property after a connection has been established will result in an exception being thrown. This property should only be set after setting the **Secure** property to **true** and before calling the **Accept** or **Connect** methods.

In some cases, a server may only accept a secure connection if the TLS v1 protocol is specified. If the security protocol is not compatible with the server, then the connection will fail with an error indicating that the class is unable to establish a security context for the session. In this case, try assigning the property to **protocolTLS1** and attempt the connection again.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Status Property

Gets a value which specifies the current status of the socket.

[Visual Basic]

Public **ReadOnly** **Property** **Status** As [SocketStatus](#)

[C#]

public [SocketWrench.SocketStatus](#) **Status** {get;}

Property Value

A [SocketStatus](#) enumeration value which specifies the current socket status.

Remarks

The **Status** property returns the current status of the socket. This property should be checked on blocking sockets to determine if the socket is in use before taking some action.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.ThreadModel Property

Gets and sets a value which specifies the threading model for the class instance.

[Visual Basic]

```
Public Property ThreadModel As ThreadingModel
```

[C#]

```
public SocketWrench.ThreadingModel ThreadModel {get; set;}
```

Property Value

Returns one or more [ThreadingModel](#) enumeration value which specifies the threading model for the client. The default value for this property is **ThreadingModel.SingleThread**.

Remarks

The **ThreadModel** property specifies the threading model that is used by the class instance when a connection is established. The default value for this property is **ThreadingModel.SingleThread**, which specifies that only the thread that established the connection should be permitted to invoke methods. It is important to note that this threading model does not limit the application to a single thread of execution. When a session is established using the **Connect** method, that session is attached to the thread that created it. From that point on, until the session is terminated, only the owner may invoke methods in that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Setting this property to **ThreadingModel.FreeThread** disables certain internal safety checks that are performed by the class and may result in unexpected behavior unless you ensure that access to the class instance is synchronized across multiple threads. The application must ensure that no two threads will attempt to invoke a blocking method at the same time. In other words, if one thread invokes a method, the application must ensure that another thread will not attempt to invoke any other method at the same time using the same instance of the class.

Changing the value of this property will not affect an active client session. The threading model must be specified prior to invoking the **Connect** method.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [AttachThread Method](#) | [ThreadingModel Enumeration](#) | [ThreadModel Attribute](#)

SocketWrench.ThrowError Property

Gets and sets a value which specifies if method calls should throw exceptions when an error occurs.

[Visual Basic]

Public Property ThrowError As Boolean

[C#]

public bool ThrowError {get; set;}

Property Value

Returns **true** if method calls will generate exceptions when an error occurs; otherwise returns **false**. The default value is **false**.

Remarks

Error handling for when calling class methods can be done in either of two different styles, according to the value of this property.

If the **ThrowError** property is set to **false**, the application should check the return value of any method that is used, and report errors based upon the documented value of the return code. It is the responsibility of the application to interpret the error code, if it is desired to explain the error in addition to reporting it. This is the default behavior.

If the **ThrowError** property is set to **true**, then exceptions will be generated whenever a method call fails. The program must be written to catch these exceptions and take the appropriate action when an error occurs. Failure to handle an exception will cause the program to terminate abnormally.

Note that if an error occurs while a property is being read or modified, an exception will be raised regardless of the value of the **ThrowError** property.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Timeout Property

Gets and sets a value which specifies a timeout period in seconds.

[Visual Basic]

Public Property Timeout As Integer

[C#]

public int Timeout {get; set;}

Property Value

An integer value which specifies a timeout period in seconds.

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking socket operation fails and returns an error.

The timeout period is only used when the socket is in blocking mode. Although this property can be changed when the socket is in non-blocking mode, the value will be ignored until the socket is returned to blocking mode.

For most applications it is recommended the timeout period be set between 10 and 20 seconds.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Trace Property

Gets and sets a value which indicates if network function logging is enabled.

[Visual Basic]

```
Public Property Trace As Boolean
```

[C#]

```
public bool Trace {get; set;}
```

Property Value

Returns **true** if network function tracing is enabled; otherwise returns **false**. The default value is **false**.

Remarks

The **Trace** property is used to enable (or disable) the tracing of network function calls. When enabled, each function call is logged to a file, including the function parameters, return value and error code if applicable. This facility can be enabled and disabled at run time, and the trace log file can be specified by setting the **TraceFile** property. All function calls that are being logged are appended to the trace file, if it exists. If no trace file exists when tracing is enabled, the trace file is created.

The tracing facility is available in all of the SocketTools networking classes and is enabled or disabled for an entire process. This means that once trace logging is enabled for a given component, all of the function calls made by the process using any of the SocketTools classes will be logged. For example, if you have an application using both the File Transfer Protocol and Post Office Protocol classes, and you set the **Trace** property to **true**, function calls made by both classes will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all classes used by the process.

If trace logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which multiple processes are being traced or the logfile is fairly large. Since logfiles can grow very quickly, even with modest applications, it is recommended that you delete the file when it is no longer needed.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.TraceFile Property

Gets and sets a value which specifies the name of the network function tracing logfile.

[Visual Basic]

```
Public Property TraceFile As String
```

[C#]

```
public string TraceFile {get; set;}
```

Property Value

A string which specifies the name of the file.

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when network function tracing is enabled. If this property is set to an empty string (the default value), then a file named **SocketTools.log** is created in the system's temporary directory. If no temporary directory exists, then the file is created in the current working directory.

If the file exists, the trace output is appended to the file, otherwise the file is created. Since network function tracing is enabled per-process, the trace file is shared by all instances of the class being used. If multiple class instances have tracing enabled, the **TraceFile** property should be set to the same value for each instance. Since trace files can grow very quickly, even with modest applications, it is recommended that you delete the file when it is no longer needed.

The trace file has the following format:

```
MyApp INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0 MyApp WRN:  
connect(46, 192.0.0.1:1234, 16) returned -1 [10035] MyApp ERR: accept(46, NULL,  
0x0) returned -1 [10038]
```

The first column contains the name of the process that is being traced. The second column identifies if the trace record is reporting information, a warning, or an error. What follows is the name of the function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record (the value is placed inside brackets).

If parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a pointer (a memory address), it is recorded as a hexadecimal value preceded with "0x". A special type of pointer, called a null pointer, is recorded as NULL. Those functions which expect socket addresses are displayed in the following format:

```
aa.bb.cc.dd:nnnn
```

The first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. Note that in the second line of the above example, the class is attempting to connect to a system with the IP address 192.0.0.1 on port 1234.

Note that if the specified file cannot be created, or the user does not have permission to modify an existing file, the error is silently ignored and no trace output will be generated.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.TraceFlags Property

Gets and sets a value which specifies the network function tracing flags.

[Visual Basic]

Public Property TraceFlags As [TraceOptions](#)

[C#]

public [SocketWrench.TraceOptions](#) TraceFlags {get; set;}

Property Value

A [TraceOptions](#) enumeration which specifies the amount of detail written to the trace logfile.

Remarks

The **TraceFlags** property is used to specify the type of information written to the trace file when network function tracing is enabled.

Because network function tracing is enabled per-process, the trace flags are shared by all instances of the class being used. If multiple class instances have tracing enabled, the **TraceFlags** property should be set to the same value for each instance. Changing the trace flags for any one instance of the class will affect the logging performed for all SocketTools classes used by the application.

Warnings are generated when a non-fatal error is returned by a network function. For example, if data is being written and the error **OperationWouldBlock** occurs, a warning is generated because the application simply needs to attempt to write the data at a later time.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Urgent Property

Gets and sets a value which specifies if urgent data will be read or written.

[Visual Basic]

Public Property Urgent As Boolean

[C#]

public bool Urgent {get; set;}

Property Value

Returns **true** if urgent data will be read or written; otherwise returns **false**. The default value is **false**.

Remarks

The **Urgent** property affects how the **Read** and **Write** methods receive and transmit data to the remote host. If set to a value of **true**, urgent (out-of-band) data will be read or written. The property value will automatically be reset to a value of **false** after the data has been read or written.

It is important to note that all systems may support more than one byte of urgent data if the data is not being received in-line. Refer to the **InLine** property for additional information. This property should only be set to **true** if required by the application and the implications of doing so are understood.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Version Property

Gets a value which returns the current version of the SocketWrench class library.

[Visual Basic]

Public ReadOnly Property Version As String

[C#]

public string Version {get;}

Property Value

A string which specifies the version of the class library.

Remarks

The **Version** property returns a string which identifies the current version and build of the SocketWrench class library. This value can be used by an application for validation and debugging purposes.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench Methods

The methods of the **SocketWrench** class are listed below. For a complete list of **SocketWrench** class members, see the [SocketWrench Members](#) topic.

Public Instance Methods

Abort	Abort the connection with a remote host.
Accept	Overloaded. Accepts a client connection on a listening socket, specifying a timeout period and one or more socket options.
AttachThread	Attach an instance of the class to the current thread
Bind	Overloaded. Bind the socket to the specified local address and port number.
Cancel	Cancel the current blocking socket operation.
Connect	Overloaded. Establish a connection with a remote host.
ConnectUrl	Overloaded. Establish a connection with a remote host using a URL.
Disconnect	Terminate the connection with a remote host.
Dispose	Overloaded. Releases all resources used by SocketWrench .
Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
Flush	Flush the contents of the send and receive socket buffers.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
Initialize	Overloaded. Initialize an instance of the SocketWrench class.
Listen	Overloaded. Listen for incoming client connections, specifying the local network address, port number and connection backlog.
Peek	Overloaded. Read data from the socket and store it in a byte array, but do not remove the data from the socket buffers.
Read	Overloaded. Read data from the socket and store it in a byte array.
ReadFrom	Overloaded. Read data from the socket and store it in a byte array.
ReadLine	Overloaded. Read up to a line of data from the

	socket and return it in a string buffer.
ReadStream	Overloaded. Read a data stream from the socket and store it in the specified byte array.
Reject	Rejects a connection request from a remote host.
Reset	Reset the internal state of the object, resetting all properties to their default values.
Resolve	Resolves a host name to a host IP address.
Shutdown	Overloaded. Disable sending or receiving data on the socket.
StoreStream	Overloaded. Reads a data stream from the socket and stores it in the specified file.
ToString (inherited from Object)	Returns a String that represents the current Object.
Uninitialize	Uninitialize the class library and release any resources allocated for the current thread.
Write	Overloaded. Write one or more bytes of data to the socket.
WriteLine	Overloaded. Send a line of text to the remote host, terminated by a carriage-return and linefeed.
WriteStream	Overloaded. Write a stream of bytes to the socket.
WriteTo	Overloaded. Write one or more bytes of data to the socket.

Protected Instance Methods

 Dispose	Overloaded. Releases the unmanaged resources allocated by the SocketWrench class and optionally releases the managed resources.
 Finalize	Destroys an instance of the class, releasing the resources allocated for the session and unloading the networking library.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Abort Method

Abort the connection with a remote host.

[Visual Basic]

```
Public Sub Abort()
```

[C#]

```
public void Abort();
```

Remarks

The **Abort** method immediately closes the socket, without waiting for any remaining data to be written out. This method should only be used when the connection must be closed immediately. If this method is used, the remote host will see the connection as being terminated abnormally.

It is recommended that applications using the **Disconnect** method unless it is absolutely necessary to terminate the connection and immediately release the socket handle.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [Disconnect Method](#)

SocketWrench.Accept Method

Accepts a client connection on a listening socket.

Overload List

Accepts a client connection on a listening socket.

```
public bool Accept(int);
```

Accepts a client connection on a listening socket, specifying one or more socket options.

```
public bool Accept(int,SocketOptions);
```

Accepts a client connection on a listening socket, specifying a timeout period and one or more socket options.

```
public bool Accept(int,int,SocketOptions);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [Listen Method](#)

SocketWrench.Accept Method (Int32)

Accepts a client connection on a listening socket.

[Visual Basic]

```
Overloads Public Function Accept( _  
    ByVal handle As Integer _  
) As Boolean
```

[C#]

```
public bool Accept(  
    int handle  
);
```

Parameters

handle

The socket identifier of a listening socket. If the object that invokes this method is not the listening socket, then the listening socket may continue to listen for incoming connections. If the object of a listening socket invokes this method with its own handle, then it ceases to listen, and no other host can establish a connection with the application.

Return Value

A boolean value which specifies if the client connection has been accepted. If the method returns **true**, the connection has been accepted and the application may send and receive data with the remote host. If this method returns **false**, the connection could not be accepted and the application should check the value of the **LastError** property to determine the cause of the failure.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Accept Overload List](#) | [Listen Method](#)

SocketWrench.Accept Method (Int32, Int32, SocketOptions)

Accepts a client connection on a listening socket, specifying a timeout period and one or more socket options.

[Visual Basic]

```
Overloads Public Function Accept( _  
    ByVal handle As Integer, _  
    ByVal timeout As Integer, _  
    ByVal options As SocketOptions _  
) As Boolean
```

[C#]

```
public bool Accept(  
    int handle,  
    int timeout,  
    SocketOptions options  
);
```

Parameters

handle

The socket identifier of a listening socket. If the object that invokes this method is not the listening socket, then the listening socket may continue to listen for incoming connections. If the object of a listening socket invokes this method with its own handle, then it ceases to listen, and no other host can establish a connection with the application.

timeout

Specifies the number of seconds that the method will wait for a client connection to be established on the listening socket. This value only has meaning for a blocking socket.

options

One or more of the [SocketOptions](#) enumeration flags.

Return Value

A boolean value which specifies if the client connection has been accepted. If the method returns **true**, the connection has been accepted and the application may send and receive data with the remote host. If this method returns **false**, the connection could not be accepted and the application should check the value of the [LastError](#) property to determine the cause of the failure.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Accept Overload List](#) | [Listen Method](#)

SocketWrench.Accept Method (Int32, SocketOptions)

Accepts a client connection on a listening socket, specifying one or more socket options.

[Visual Basic]

```
Overloads Public Function Accept( _  
    ByVal handle As Integer, _  
    ByVal options As SocketOptions _  
) As Boolean
```

[C#]

```
public bool Accept(  
    int handle,  
    SocketOptions options  
);
```

Parameters

handle

The socket identifier of a listening socket. If the object that invokes this method is not the listening socket, then the listening socket may continue to listen for incoming connections. If the object of a listening socket invokes this method with its own handle, then it ceases to listen, and no other host can establish a connection with the application.

options

One or more of the [SocketOptions](#) enumeration flags.

Return Value

A boolean value which specifies if the client connection has been accepted. If the method returns **true**, the connection has been accepted and the application may send and receive data with the remote host. If this method returns **false**, the connection could not be accepted and the application should check the value of the **LastError** property to determine the cause of the failure.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Accept Overload List](#) | [Listen Method](#)

SocketWrench.AttachThread Method

Attach an instance of the class to the current thread

[Visual Basic]

```
Public Function AttachThread() As Boolean
```

[C#]

```
public bool AttachThread();
```

Return Value

A boolean value which specifies if the socket could be attached to the current thread. If this method returns **false**, the socket could not be attached to the thread and the application should check the value of the [LastError](#) property to determine the cause of the failure.

Remarks

When an instance of the class is created it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that instance, an error is returned. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in an application to create an instance of the class, establish a connection and then pass that instance to another worker thread. The **AttachThread** method can be used to change the ownership of the class instance to the new worker thread.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored by the original thread. Under no circumstances should **AttachThread** be used to forcibly destroy an instance of a class allocated by another thread while a blocking operation is in progress. To cancel a blocking operation, use the **Cancel** method and then delete the class instance after the blocking function exits and control is returned to the current thread.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Bind Method

Bind the socket to the specified local address and port number.

Overload List

Bind the socket to the specified local address and port number.

```
public bool Bind(string,int);
```

Bind the socket to the specified local address and port number.

```
public bool Bind(string,int,SocketOptions);
```

Bind the socket to the specified local address and port number.

```
public bool Bind(string,int,SocketProtocol);
```

Bind the socket to the specified local address and port number.

```
public bool Bind(string,int,SocketProtocol,SocketOptions);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Bind Method (String, Int32)

Bind the socket to the specified local address and port number.

[Visual Basic]

```
Overloads Public Function Bind( _  
    ByVal LocalAddress As String, _  
    ByVal LocalPort As Integer _  
) As Boolean
```

[C#]

```
public bool Bind(  
    string LocalAddress,  
    int LocalPort  
);
```

Parameters

localAddress

A string which specifies the local Internet address that the socket should be bound to. To bind to any valid network interface on the local system, specify the address 0.0.0.0. Applications should only specify a particular address if it is absolutely necessary. In most cases a local address is not required when establishing a client connection.

localPort

An integer value which specifies a local port number that the socket should be bound to. To bind to any available port number, specify a port number of 0. Applications should only specify a particular port number if it is absolutely necessary. The maximum valid port number is 65535.

Return Value

A boolean value which specifies if the socket could be bound to the specified address. If this method returns **false**, the socket could not be bound to the address and the application should check the value of the [LastError](#) property to determine the cause of the failure.

Remarks

The **Bind** method is used to specify the local address and port number that a socket will be bound to when it is created. When this method is called with **Datagram** as the specified protocol, it will immediately create the datagram socket and bind it to the given address.

When this method is called with **Stream** as the specified protocol, creation of the socket is deferred until the **Connect** method is called. For stream sockets, this method will set the local address, port number and default options used when the socket is actually created.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Bind Overload List](#)

SocketWrench.Bind Method (String, Int32, SocketOptions)

Bind the socket to the specified local address and port number.

[Visual Basic]

```
Overloads Public Function Bind( _  
    ByVal LocalAddress As String, _  
    ByVal LocalPort As Integer, _  
    ByVal options As SocketOptions _  
) As Boolean
```

[C#]

```
public bool Bind(  
    string LocalAddress,  
    int LocalPort,  
    SocketOptions options  
);
```

Parameters

localAddress

A string which specifies the local Internet address that the socket should be bound to. To bind to any valid network interface on the local system, specify the address 0.0.0.0. Applications should only specify a particular address if it is absolutely necessary. In most cases a local address is not required when establishing a client connection.

localPort

An integer value which specifies a local port number that the socket should be bound to. To bind to any available port number, specify a port number of 0. Applications should only specify a particular port number if it is absolutely necessary. The maximum valid port number is 65535.

options

One or more of the [SocketOptions](#) enumeration flags.

Return Value

A boolean value which specifies if the socket could be bound to the specified address. If this method returns **false**, the socket could not be bound to the address and the application should check the value of the [LastError](#) property to determine the cause of the failure.

Remarks

The **Bind** method is used to specify the local address and port number that a socket will be bound to when it is created. When this method is called with **Datagram** as the specified protocol, it will immediately create the datagram socket and bind it to the given address.

When this method is called with **Stream** as the specified protocol, creation of the socket is deferred until the **Connect** method is called. For stream sockets, this method will set the local address, port number and default options used when the socket is actually created.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Bind Overload List](#)

SocketWrench.Bind Method (String, Int32, SocketProtocol)

Bind the socket to the specified local address and port number.

[Visual Basic]

```
Overloads Public Function Bind( _  
    ByVal LocalAddress As String, _  
    ByVal LocalPort As Integer, _  
    ByVal protocol As SocketProtocol _  
) As Boolean
```

[C#]

```
public bool Bind(  
    string LocalAddress,  
    int LocalPort,  
    SocketProtocol protocol  
);
```

Parameters

localAddress

A string which specifies the local Internet address that the socket should be bound to. To bind to any valid network interface on the local system, specify the address 0.0.0.0. Applications should only specify a particular address if it is absolutely necessary. In most cases a local address is not required when establishing a client connection.

localPort

An integer value which specifies a local port number that the socket should be bound to. To bind to any available port number, specify a port number of 0. Applications should only specify a particular port number if it is absolutely necessary. The maximum valid port number is 65535.

protocol

One of the [SocketProtocol](#) enumeration values which specify the type of socket to be created.

Return Value

A boolean value which specifies if the socket could be bound to the specified address. If this method returns **false**, the socket could not be bound to the address and the application should check the value of the [LastError](#) property to determine the cause of the failure.

Remarks

The **Bind** method is used to specify the local address and port number that a socket will be bound to when it is created. When this method is called with **Datagram** as the specified protocol, it will immediately create the datagram socket and bind it to the given address.

When this method is called with **Stream** as the specified protocol, creation of the socket is deferred until the **Connect** method is called. For stream sockets, this method will set the local address, port number and default options used when the socket is actually created.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Bind Overload List](#)

SocketWrench.Bind Method (String, Int32, SocketProtocol, SocketOptions)

Bind the socket to the specified local address and port number.

[Visual Basic]

```
Overloads Public Function Bind( _  
    ByVal LocalAddress As String, _  
    ByVal LocalPort As Integer, _  
    ByVal protocol As SocketProtocol, _  
    ByVal options As SocketOptions _  
) As Boolean
```

[C#]

```
public bool Bind(  
    string LocalAddress,  
    int LocalPort,  
    SocketProtocol protocol,  
    SocketOptions options  
);
```

Parameters

localAddress

A string which specifies the local Internet address that the socket should be bound to. To bind to any valid network interface on the local system, specify the address 0.0.0.0. Applications should only specify a particular address if it is absolutely necessary. In most cases a local address is not required when establishing a client connection.

localPort

An integer value which specifies a local port number that the socket should be bound to. To bind to any available port number, specify a port number of 0. Applications should only specify a particular port number if it is absolutely necessary. The maximum valid port number is 65535.

protocol

One of the [SocketProtocol](#) enumeration values which specify the type of socket to be created.

options

One or more of the [SocketOptions](#) enumeration flags.

Return Value

A boolean value which specifies if the socket could be bound to the specified address. If this method returns **false**, the socket could not be bound to the address and the application should check the value of the [LastError](#) property to determine the cause of the failure.

Remarks

The **Bind** method is used to specify the local address and port number that a socket will be bound to when it is created. When this method is called with **Datagram** as the specified protocol, it will immediately create the datagram socket and bind it to the given address.

When this method is called with **Stream** as the specified protocol, creation of the socket is deferred until the **Connect** method is called. For stream sockets, this method will set the local address, port number and default options used when the socket is actually created.

See Also

SocketWrench.Cancel Method

Cancel the current blocking socket operation.

[Visual Basic]

```
Public Sub Cancel()
```

[C#]

```
public void Cancel();
```

Remarks

When the **Cancel** method is called, the blocking socket operation will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other blocking function. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Connect Method

Establish a connection with a remote host.

Overload List

Establish a connection with a remote host.

```
public bool Connect();
```

Establish a connection with a remote host.

```
public bool Connect(string,int);
```

Establish a connection with a remote host.

```
public bool Connect(string,int,SocketOptions,int);
```

Establish a connection with a remote host.

```
public bool Connect(string,int,SocketProtocol,int);
```

Establish a connection with a remote host.

```
public bool Connect(string,int,SocketProtocol,int,SocketOptions);
```

Establish a connection with a remote host.

```
public bool Connect(string,int,SocketProtocol,int,SocketOptions,string,int);
```

Establish a connection with a remote host.

```
public bool Connect(string,int,int);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Connect Method (String, Int32)

Establish a connection with a remote host.

[Visual Basic]

```
Overloads Public Function Connect( _  
    ByVal hostName As String, _  
    ByVal hostPort As Integer _  
) As Boolean
```

[C#]

```
public bool Connect(  
    string hostName,  
    int hostPort  
);
```

Parameters

hostName

A string which specifies the remote host to establish a connection with. This may specify a host name or an Internet address in dot-notation.

hostPort

An integer which specifies the port number for the connection. This value must be greater than zero and the maximum valid port number is 65535.

Return Value

A boolean value which specifies if the connection has been established. If the socket is in blocking mode, a return value of **true** indicates that the connection has completed and the application may send and receive data from the remote host. If the socket is in non-blocking mode, a return value of **true** indicates that the socket has been successfully created and the connection is in progress. When the connection has completed, the [OnConnect](#) event will be fired. If this method returns **false**, the connection could not be established and the application should check the value of the [LastError](#) property to determine the cause of the failure.

Remarks

This method will use the value of the [Protocol](#) property to determine which protocol is used to establish the connection. By default, the **Stream** protocol will be used.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Connect Overload List](#)

SocketWrench.Connect Method (String, Int32, Int32)

Establish a connection with a remote host.

[Visual Basic]

```
Overloads Public Function Connect( _  
    ByVal hostName As String, _  
    ByVal hostPort As Integer, _  
    ByVal timeout As Integer _  
) As Boolean
```

[C#]

```
public bool Connect(  
    string hostName,  
    int hostPort,  
    int timeout  
);
```

Parameters

hostName

A string which specifies the remote host to establish a connection with. This may specify a host name or an Internet address in dot-notation.

hostPort

An integer which specifies the port number for the connection. This value must be greater than zero and the maximum valid port number is 65535.

timeout

An integer value that specifies the number of seconds that the method will wait for the connection to complete before failing the operation and returning to the caller. This value is only meaningful for blocking sockets.

Return Value

A boolean value which specifies if the connection has been established. If the socket is in blocking mode, a return value of **true** indicates that the connection has completed and the application may send and receive data from the remote host. If the socket is in non-blocking mode, a return value of **true** indicates that the socket has been successfully created and the connection is in progress.

When a non-blocking connection has completed, the [OnConnect](#) event will be fired. If this method returns **false**, the connection could not be established and the application should check the value of the [LastError](#) property to determine the cause of the failure.

Remarks

This method will use the value of the [Protocol](#) property to determine which protocol is used to establish the connection. By default, the **Stream** protocol will be used.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Connect Overload List](#)

SocketWrench.Connect Method (String, Int32, SocketOptions, Int32)

Establish a connection with a remote host.

[Visual Basic]

```
Overloads Public Function Connect( _  
    ByVal hostName As String, _  
    ByVal hostPort As Integer, _  
    ByVal options As SocketOptions, _  
    ByVal timeout As Integer _  
) As Boolean
```

[C#]

```
public bool Connect(  
    string hostName,  
    int hostPort,  
    SocketOptions options,  
    int timeout  
);
```

Parameters

hostName

A string which specifies the remote host to establish a connection with. This may specify a host name or an Internet address in dot-notation.

hostPort

An integer which specifies the port number for the connection. This value must be greater than zero and the maximum valid port number is 65535.

options

One or more of the [SocketOptions](#) enumeration flags.

timeout

An integer value that specifies the number of seconds that the method will wait for the connection to complete before failing the operation and returning to the caller. This value is only meaningful for blocking sockets.

Return Value

A boolean value which specifies if the connection has been established. If the socket is in blocking mode, a return value of **true** indicates that the connection has completed and the application may send and receive data from the remote host. If the socket is in non-blocking mode, a return value of **true** indicates that the socket has been successfully created and the connection is in progress.

When a non-blocking connection has completed, the [OnConnect](#) event will be fired. If this method returns **false**, the connection could not be established and the application should check the value of the [LastError](#) property to determine the cause of the failure.

Remarks

This method will use the value of the [Protocol](#) property to determine which protocol is used to establish the connection. By default, the **Stream** protocol will be used.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Connect Overload List](#)

SocketWrench.Connect Method (String, Int32, SocketProtocol, Int32)

Establish a connection with a remote host.

[Visual Basic]

```
Overloads Public Function Connect( _  
    ByVal hostName As String, _  
    ByVal hostPort As Integer, _  
    ByVal protocol As SocketProtocol, _  
    ByVal timeout As Integer _  
) As Boolean
```

[C#]

```
public bool Connect(  
    string hostName,  
    int hostPort,  
    SocketProtocol protocol,  
    int timeout  
);
```

Parameters

hostName

A string which specifies the remote host to establish a connection with. This may specify a host name or an Internet address in dot-notation.

hostPort

An integer which specifies the port number for the connection. This value must be greater than zero and the maximum valid port number is 65535.

protocol

One of the [SocketProtocol](#) enumeration values which specify the type of socket to be created.

timeout

An integer value that specifies the number of seconds that the method will wait for the connection to complete before failing the operation and returning to the caller. This value is only meaningful for blocking sockets.

Return Value

A boolean value which specifies if the connection has been established. If the socket is in blocking mode, a return value of **true** indicates that the connection has completed and the application may send and receive data from the remote host. If the socket is in non-blocking mode, a return value of **true** indicates that the socket has been successfully created and the connection is in progress.

When a non-blocking connection has completed, the [OnConnect](#) event will be fired. If this method returns **false**, the connection could not be established and the application should check the value of the [LastError](#) property to determine the cause of the failure.

Remarks

When this method is called with **Datagram** as the specified protocol, it does not actually establish a connection. Instead, it simply establishes a default destination address and port that is used with subsequent calls to the **Read** and **Write** methods.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Connect Overload List](#)

SocketWrench.Connect Method (String, Int32, SocketProtocol, Int32, SocketOptions)

Establish a connection with a remote host.

[Visual Basic]

```
Overloads Public Function Connect( _  
    ByVal hostName As String, _  
    ByVal hostPort As Integer, _  
    ByVal protocol As SocketProtocol, _  
    ByVal timeout As Integer, _  
    ByVal options As SocketOptions _  
) As Boolean
```

[C#]

```
public bool Connect(  
    string hostName,  
    int hostPort,  
    SocketProtocol protocol,  
    int timeout,  
    SocketOptions options  
);
```

Parameters

hostName

A string which specifies the remote host to establish a connection with. This may specify a host name or an Internet address in dot-notation.

hostPort

An integer which specifies the port number for the connection. This value must be greater than zero and the maximum valid port number is 65535.

protocol

One of the [SocketProtocol](#) enumeration values which specify the type of socket to be created.

timeout

An integer value that specifies the number of seconds that the method will wait for the connection to complete before failing the operation and returning to the caller. This value is only meaningful for blocking sockets.

options

One or more of the [SocketOptions](#) enumeration flags.

Return Value

A boolean value which specifies if the connection has been established. If the socket is in blocking mode, a return value of **true** indicates that the connection has completed and the application may send and receive data from the remote host. If the socket is in non-blocking mode, a return value of **true** indicates that the socket has been successfully created and the connection is in progress.

When a non-blocking connection has completed, the [OnConnect](#) event will be fired. If this method returns **false**, the connection could not be established and the application should check the value of the [LastError](#) property to determine the cause of the failure.

Remarks

When this method is called with **Datagram** as the specified protocol, it does not actually establish a connection. Instead, it simply establishes a default destination address and port that is used with subsequent

calls to the **Read** and **Write** methods.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Connect Overload List](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketWrench.Connect Method (String, Int32, SocketProtocol, Int32, SocketOptions, String, Int32)

Establish a connection with a remote host.

[Visual Basic]

```
Overloads Public Function Connect( _  
    ByVal hostName As String, _  
    ByVal hostPort As Integer, _  
    ByVal protocol As SocketProtocol, _  
    ByVal timeout As Integer, _  
    ByVal options As SocketOptions, _  
    ByVal localAddress As String, _  
    ByVal localPort As Integer _  
) As Boolean
```

[C#]

```
public bool Connect(  
    string hostName,  
    int hostPort,  
    SocketProtocol protocol,  
    int timeout,  
    SocketOptions options,  
    string localAddress,  
    int localPort  
);
```

Parameters

hostName

A string which specifies the remote host to establish a connection with. This may specify a host name or an Internet address in dot-notation.

hostPort

An integer which specifies the port number for the connection. This value must be greater than zero and the maximum valid port number is 65535.

protocol

One of the [SocketProtocol](#) enumeration values which specify the type of socket to be created.

timeout

An integer value that specifies the number of seconds that the method will wait for the connection to complete before failing the operation and returning to the caller. This value is only meaningful for blocking sockets.

options

One or more of the [SocketOptions](#) enumeration flags.

localAddress

A string which specifies the local Internet address that the socket should be bound to. To bind to any valid network interface on the local system, specify the address 0.0.0.0. Applications should only specify a particular address if it is absolutely necessary. In most cases a local address is not required when establishing a client connection.

localPort

An integer value which specifies a local port number that the socket should be bound to. To bind to any available port number, specify a port number of 0. Applications should only specify a particular port number if it is absolutely necessary. The maximum valid port number is 65535.

Return Value

A boolean value which specifies if the connection has been established. If the socket is in blocking mode, a return value of **true** indicates that the connection has completed and the application may send and receive data from the remote host. If the socket is in non-blocking mode, a return value of **true** indicates that the socket has been successfully created and the connection is in progress.

When a non-blocking connection has completed, the [OnConnect](#) event will be fired. If this method returns **false**, the connection could not be established and the application should check the value of the [LastError](#) property to determine the cause of the failure.

Remarks

When this method is called with **Datagram** as the specified protocol, it does not actually establish a connection. Instead, it simply establishes a default destination address and port that is used with subsequent calls to the [Read](#) and [Write](#) methods.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Connect Overload List](#)

SocketWrench.ConnectUrl Method

Establish a connection with a remote host using a URL.

Overload List

Establish a connection with a remote host using a URL.

```
public bool ConnectUrl(string);
```

Establish a connection with a remote host using a URL.

```
public bool ConnectUrl(string,int,SocketOptions);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [Connect Method](#)

SocketWrench.ConnectUrl Method (String)

Establish a connection with a remote host using a URL.

[Visual Basic]

```
Overloads Public Function ConnectUrl( _  
    ByVal hostUrl As String _  
) As Boolean
```

[C#]

```
public bool ConnectUrl(  
    string hostUrl  
);
```

Parameters

hostUrl

A string which specifies the URL used to establish a connection. This parameter cannot be null or an empty string.

Return Value

A boolean value which specifies if the connection has been established. If the socket is in blocking mode, a return value of **true** indicates that the connection has completed and the application may send and receive data from the remote host. If the socket is in non-blocking mode, a return value of **true** indicates that the socket has been successfully created and the connection is in progress.

When a non-blocking connection has completed, the [OnConnect](#) event will be fired. If this method returns **false**, the connection could not be established and the application should check the value of the [LastError](#) property to determine the cause of the failure.

Remarks

The ConnectUrl method provides a simplified interface which can be used to establish a connection using a URL. This method can only be used to establish connections using TCP and does not currently support the use of URLs to connect with a service which uses UDP. The general format of the URL should look like this:

```
[scheme]:// [[username : password] @] hostname [:port] / [path;parameters  
...]
```

This method recognizes most standard URI schemes which use this format. The host name and port number specified in the URL will be used to establish a connection and the remaining information will be discarded. If the URL does not explicitly specify a port number, the default port number associated with the scheme will be used as the default value. For example, consider the following:

```
https://www.example.com
```

In this example, there is no port number specified; instead, the default port for the **https://** scheme would be used, which is port 443. The host name **www.example.com** would be resolved into an IP address and the connection established on port 443. This method will also recognize a simpler format which only includes the host name and port number without a URI scheme, such as:

```
www.example.com:443
```

When used in this way, the port number must always be provided. Without a URI scheme or an explicit port number, the method cannot determine what port number should be used when establishing the connection.

The same also applies if a custom, non-standard URI scheme is provided which is not recognized.

If the URI scheme specifies a secure protocol which requires implicit TLS, this function will automatically enable the **Secure** option. For example, providing a URL which uses the **https://** scheme will automatically enable a secure connection. If a URI scheme is used in conjunction with a port number associated with a secure service, security will also be enabled for that connection. For example:

http://www.example.com:443

The standard **http://** scheme is used which does not indicate a secure connection. However, since port 443 is the standard port designated for a secure HTTP connection, a secure connection will be enabled by default, even if not been specified by the caller. Alternatively, if a custom port number is specified in the URL or the scheme is not recognized as one which requires implicit TLS, security options will not be automatically enabled by default for the connection.

The host name portion of the URL can be specified as either a domain name or an IP address. Because an IPv6 address can contain colon characters, you must enclose the entire address in bracket [] characters. If this is not done, this method will return an error indicating the port number is invalid. For example, the URL **https://[2001:db8:0:0:1::128]/** uses an IPv6 host address and this would be considered valid. Without the brackets, this URL would not be accepted.

Important: The URL provided to this method will only be used to establish a connection with a server. This is a general purpose method which does not enable support for any particular application protocol and all implementation details are the responsibility of your application. If you require higher-level support for a specific Internet protocol, the SocketTools .NET Edition provides comprehensive collection of higher-level classes which can be used to access those services.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling `ConnectUrl` in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ConnectUrl Overload List](#) | [Connect Method](#)

SocketWrench.ConnectUrl Method (String, Int32, SocketOptions)

Establish a connection with a remote host using a URL.

[Visual Basic]

```
Overloads Public Function ConnectUrl( _  
    ByVal hostUrl As String, _  
    ByVal timeout As Integer, _  
    ByVal options As SocketOptions _  
) As Boolean
```

[C#]

```
public bool ConnectUrl(  
    string hostUrl,  
    int timeout,  
    SocketOptions options  
);
```

Parameters

hostUrl

A string which specifies the URL used to establish a connection. This parameter cannot be null or an empty string.

timeout

An integer value that specifies the number of seconds that the method will wait for the connection to complete before failing the operation and returning to the caller. This value is only meaningful for blocking sockets.

options

One or more of the [SocketOptions](#) enumeration flags.

Return Value

A boolean value which specifies if the connection has been established. If the socket is in blocking mode, a return value of **true** indicates that the connection has completed and the application may send and receive data from the remote host. If the socket is in non-blocking mode, a return value of **true** indicates that the socket has been successfully created and the connection is in progress.

When a non-blocking connection has completed, the [OnConnect](#) event will be fired. If this method returns **false**, the connection could not be established and the application should check the value of the [LastError](#) property to determine the cause of the failure.

Remarks

The ConnectUrl method provides a simplified interface which can be used to establish a connection using a URL. This method can only be used to establish connections using TCP and does not currently support the use of URLs to connect with a service which uses UDP. The general format of the URL should look like this:

```
[scheme]:// [[username : password] @] hostname [:port] / [path;parameters  
...]
```

This method recognizes most standard URI schemes which use this format. The host name and port number specified in the URL will be used to establish a connection and the remaining information will be discarded. If the URL does not explicitly specify a port number, the default port number associated with the scheme will be used as the default value. For example, consider the following:

https://www.example.com

In this example, there is no port number specified; instead, the default port for the **https://** scheme would be used, which is port 443. The host name **www.example.com** would be resolved into an IP address and the connection established on port 443. This method will also recognize a simpler format which only includes the host name and port number without a URI scheme, such as:

www.example.com:443

When used in this way, the port number must always be provided. Without a URI scheme or an explicit port number, the method cannot determine what port number should be used when establishing the connection. The same also applies if a custom, non-standard URI scheme is provided which is not recognized.

If the URI scheme specifies a secure protocol which requires implicit TLS, this function will automatically enable the **Secure** option. For example, providing a URL which uses the **https://** scheme will automatically enable a secure connection. If a URI scheme is used in conjunction with a port number associated with a secure service, security will also be enabled for that connection. For example:

http://www.example.com:443

The standard **http://** scheme is used which does not indicate a secure connection. However, since port 443 is the standard port designated for a secure HTTP connection, a secure connection will be enabled by default, even if not been specified by the caller. Alternatively, if a custom port number is specified in the URL or the scheme is not recognized as one which requires implicit TLS, security options will not be automatically enabled by default for the connection.

The host name portion of the URL can be specified as either a domain name or an IP address. Because an IPv6 address can contain colon characters, you must enclose the entire address in bracket [] characters. If this is not done, this method will return an error indicating the port number is invalid. For example, the URL **https://[2001:db8:0:0:1::128]/** uses an IPv6 host address and this would be considered valid. Without the brackets, this URL would not be accepted.

Important: The URL provided to this method will only be used to establish a connection with a server. This is a general purpose method which does not enable support for any particular application protocol and all implementation details are the responsibility of your application. If you require higher-level support for a specific Internet protocol, the SocketTools .NET Edition provides comprehensive collection of higher-level classes which can be used to access those services.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling `ConnectUrl` in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ConnectUrl Overload List](#) | [Connect Method](#)

SocketWrench.Disconnect Method

Terminate the connection with a remote host.

[Visual Basic]

```
Public Sub Disconnect()
```

[C#]

```
public void Disconnect();
```

Remarks

The **Disconnect** method terminates the connection with the remote host and closes the socket handle allocated by the class. Note that the socket is not immediately released when the connection is terminated and will enter a wait state for two minutes. After the time wait period has elapsed, the socket will be released by the operating system. This is a normal safety mechanism to handle any packets that may arrive after the connection has been closed.

To immediately terminate the connection and release the socket, use the **Abort** method.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [Abort Method](#)

SocketWrench.Dispose Method

Releases all resources used by [SocketWrench](#).

Overload List

Releases all resources used by [SocketWrench](#).

```
public void Dispose();
```

Releases the unmanaged resources allocated by the [SocketWrench](#) class and optionally releases the managed resources.

```
protected virtual void Dispose(bool);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Dispose Method ()

Releases all resources used by [SocketWrench](#).

[Visual Basic]

```
NotOverridable Overloads Public Sub Dispose() _  
    Implements IDisposable.Dispose
```

[C#]

```
public void Dispose();
```

Implements

IDisposable.Dispose

Remarks

The **Dispose** method terminates any active connection and explicitly releases the resources allocated for this instance of the class. In some cases, better performance can be achieved if the programmer explicitly releases resources when they are no longer being used. The **Dispose** method provides explicit control over these resources.

Unlike the **Uninitialize** method, once the **Dispose** method has been called, that instance of the class cannot be re-initialized and you should not attempt to access class properties or invoke any methods. Note that this method can be called even if other references to the object are active.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Dispose Overload List](#)

SocketWrench.Dispose Method (Boolean)

Releases the unmanaged resources allocated by the [SocketWrench](#) class and optionally releases the managed resources.

[Visual Basic]

```
Overridable Overloads Protected Sub Dispose( _  
    ByVal disposing As Boolean _  
)
```

[C#]

```
protected virtual void Dispose(  
    bool disposing  
);
```

Parameters

disposing

A boolean value which should be specified as **true** to release both managed and unmanaged resources; **false** to release only unmanaged resources.

Remarks

The **Dispose** method terminates any active connection and explicitly releases the resources allocated for this instance of the class. In some cases, better performance can be achieved if the programmer explicitly releases resources when they are no longer being used. The **Dispose** method provides explicit control over these resources.

Unlike the **Uninitialize** method, once the **Dispose** method has been called, that instance of the class cannot be re-initialized and you should not attempt to access class properties or invoke any methods. Note that this method can be called even if other references to the object are active.

You should call **Dispose** in your derived class when you are finished using the derived class. The **Dispose** method leaves the derived class in an unusable state. After calling **Dispose**, you must release all references to the derived class and the **SocketWrench** class so the memory they were occupying can be reclaimed by garbage collection.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Dispose Overload List](#)

SocketWrench.Finalize Method

Destroys an instance of the class, releasing the resources allocated for the session and unloading the networking library.

[Visual Basic]

Overrides Protected Sub Finalize()

[C#]

protected override void Finalize();

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Initialize Method

Initialize an instance of the SocketWrench class.

Overload List

Initialize an instance of the SocketWrench class.

```
public bool Initialize();
```

Initialize an instance of the SocketWrench class.

```
public bool Initialize(string);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [Uninitialize Method](#)

SocketWrench.Initialize Method ()

Initialize an instance of the SocketWrench class.

[Visual Basic]

Overloads Public Function Initialize() As Boolean

[C#]

public bool Initialize();

Return Value

A boolean value which specifies if the class was initialized successfully.

Remarks

The Initialize method can be used to explicitly initialize an instance of the SocketWrench class, loading the networking library and allocating resources for the current thread. Typically it is not necessary to explicitly call this method because the instance of the class is initialized by the class constructor. However, if the **Uninitialize** method is called, the class must be re-initialized before any other methods are called.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Initialize Overload List](#) | [Uninitialize Method](#)

SocketWrench.Initialize Method (String)

Initialize an instance of the SocketWrench class.

[Visual Basic]

```
Overloads Public Function Initialize( _  
    ByVal LicenseKey As String _  
) As Boolean
```

[C#]

```
public bool Initialize(  
    string LicenseKey  
);
```

Return Value

A boolean value which specifies if the class was initialized successfully.

Remarks

The Initialize method can be used to explicitly initialize an instance of the SocketWrench class, loading the networking library and allocating resources for the current thread. Typically an application would define the license key as a custom attribute, however this method can be used to initialize the class directly.

The runtime license key for your copy of SocketWrench can be generated using the License Manager utility that is included with the product. Note that if you have installed an evaluation license, you will not have a runtime license key and cannot redistribute any applications which use the SocketWrench class.

Example

The following example shows how to use the Initialize method to initialize an instance of the class. This example assumes that the license key string has been defined in code.

```
SocketTools.SocketWrench socket = new SocketTools.SocketWrench();  
  
if (socket.Initialize(strLicenseKey) == false)  
{  
    MessageBox.Show(socket.LastErrorString, "Error",  
        MessageBoxButtons.OK, MessageBoxIcon.Exclamation);  
    return;  
}  
  
Dim Socket As New SocketTools.SocketWrench  
  
If Socket.Initialize(strLicenseKey) = False Then  
    MsgBox(Socket.LastErrorString, vbIconExclamation)  
    Exit Sub  
End If
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Initialize Overload List](#) | [RuntimeLicenseAttribute Class](#) | [Uninitialize Method](#)

SocketWrench.Listen Method

Listen for incoming client connections.

Overload List

Listen for incoming client connections.

```
public bool Listen();
```

Listen for incoming client connections, specifying the local port number.

```
public bool Listen(int);
```

Listen for incoming client connections, specifying the local network address and port number.

```
public bool Listen(string,int);
```

Listen for incoming client connections, specifying the local network address, port number and connection backlog.

```
public bool Listen(string,int,int);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [Blocking Property](#) | [LocalAddress Property](#) | [LocalPort Property](#) | [Accept Method](#) | [OnAccept Event](#)

SocketWrench.Listen Method ()

Listen for incoming client connections.

[Visual Basic]

Overloads Public Function Listen() As Boolean

[C#]

public bool Listen();

Return Value

A boolean value which specifies if the listening socket could be created successfully. A value of **true** indicates that a listening socket has been created. A value of **false** indicates that a listening socket could not be created using the specified address or port number and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The value of the **LocalAddress** property is used to specify the network address that will be used to listen for client connections. If the property has not been set, or is set to the address 0.0.0.0 then connections will be listened for on any valid network adapter configured on the system.

The value of the **LocalPort** property is used to specify the port number to listen for connections on.

After the listening socket has been created, the application should then call the **Accept** method to wait for a client to establish a connection. If the **Blocking** property is set to **false**, then the **OnAccept** event will fire when a client attempts to establish a connection.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Listen Overload List](#) | [Blocking Property](#) | [LocalAddress Property](#) | [LocalPort Property](#) | [Accept Method](#) | [OnAccept Event](#)

SocketWrench.Listen Method (Int32)

Listen for incoming client connections, specifying the local port number.

[Visual Basic]

```
Overloads Public Function Listen( _  
    ByVal LocalPort As Integer _  
) As Boolean
```

[C#]

```
public bool Listen(  
    int LocalPort  
);
```

Parameters

localPort

An integer argument which specifies the port number to listen for connections on. The minimum port value is 1, the maximum port value is 65535.

Return Value

A boolean value which specifies if the listening socket could be created successfully. A value of **true** indicates that a listening socket has been created. A value of **false** indicates that a listening socket could not be created using the specified address or port number and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The value of the **LocalAddress** property is used to specify the network address that will be used to listen for client connections. If the property has not been set, or is set to the address 0.0.0.0 then connections will be listened for on any valid network adapter configured on the system.

After the listening socket has been created, the application should then call the **Accept** method to wait for a client to establish a connection. If the **Blocking** property is set to **false**, then the **OnAccept** event will fire when a client attempts to establish a connection.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Listen Overload List](#) | [Blocking Property](#) | [LocalAddress Property](#) | [Accept Method](#) | [OnAccept Event](#)

SocketWrench.Listen Method (String, Int32)

Listen for incoming client connections, specifying the local network address and port number.

[Visual Basic]

```
Overloads Public Function Listen( _  
    ByVal LocalAddress As String, _  
    ByVal LocalPort As Integer _  
) As Boolean
```

[C#]

```
public bool Listen(  
    string LocalAddress,  
    int LocalPort  
);
```

Parameters

localAddress

A string argument which specifies the IP address of the network adapter that the class should use when listening for connection requests. If this argument is not specified, the class will bind to any suitable adapter on the local system. An address of 0.0.0.0 specifies that it should listen for connections on any network adapter configured on the system.

localPort

An integer argument which specifies the port number to listen for connections on. The minimum port value is 1, the maximum port value is 65535.

Return Value

A boolean value which specifies if the listening socket could be created successfully. A value of **true** indicates that a listening socket has been created. A value of **false** indicates that a listening socket could not be created using the specified address or port number and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

After the listening socket has been created, the application should then call the **Accept** method to wait for a client to establish a connection. If the **Blocking** property is set to **false**, then the **OnAccept** event will fire when a client attempts to establish a connection.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Listen Overload List](#) | [Blocking Property](#) | [Accept Method](#) | [OnAccept Event](#)

SocketWrench.Listen Method (String, Int32, Int32)

Listen for incoming client connections, specifying the local network address, port number and connection backlog.

[Visual Basic]

```
Overloads Public Function Listen( _  
    ByVal LocalAddress As String, _  
    ByVal LocalPort As Integer, _  
    ByVal backlog As Integer _  
) As Boolean
```

[C#]

```
public bool Listen(  
    string LocalAddress,  
    int LocalPort,  
    int backlog  
);
```

Parameters

localAddress

A string argument which specifies the IP address of the network adapter that the class should use when listening for connection requests. If this argument is not specified, the class will bind to any suitable adapter on the local system. An address of 0.0.0.0 specifies that it should listen for connections on any network adapter configured on the system.

localPort

An integer argument which specifies the port number to listen for connections on. The minimum port value is 1, the maximum port value is 65535.

backlog

An integer argument which specifies the maximum size of the queue used to manage pending connections to the service. If the argument is set to value which exceeds the maximum size for the underlying service provider, it will be silently adjusted to the nearest legal value.

Return Value

A boolean value which specifies if the listening socket could be created successfully. A value of **true** indicates that a listening socket has been created. A value of **false** indicates that a listening socket could not be created using the specified address or port number and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

After the listening socket has been created, the application should then call the **Accept** method to wait for a client to establish a connection. If the **Blocking** property is set to **false**, then the **OnAccept** event will fire when a client attempts to establish a connection.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Listen Overload List](#) | [Blocking Property](#) | [Accept Method](#) | [OnAccept Event](#)

SocketWrench.Peek Method

Return the number of bytes available to be read from the socket.

Overload List

Return the number of bytes available to be read from the socket.

```
public int Peek();
```

Read data from the socket and store it in a byte array, but do not remove the data from the socket buffers.

```
public int Peek(byte[]);
```

Read data from the socket and store it in a byte array, but do not remove the data from the socket buffers.

```
public int Peek(byte[],int);
```

Read data from the socket and store it in a string, but do not remove the data from the socket buffers.

```
public int Peek(ref string);
```

Read data from the socket and store it in a string, but do not remove the data from the socket buffers.

```
public int Peek(ref string,int);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Peek Method (Byte[])

Read data from the socket and store it in a byte array, but do not remove the data from the socket buffers.

[Visual Basic]

```
Overloads Public Function Peek( _  
    ByVal buffer As Byte() _  
) As Integer
```

[C#]

```
public int Peek(  
    byte[] buffer  
);
```

Parameters

buffer

A byte array that the data will be stored in.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that there is no data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Peek** method returns data that is available to read from the socket, up to the number of bytes specified. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. Using this method in a loop to poll a non-blocking socket can cause the application to become non-responsive. To determine if there is data available to be read, use the **IsReadable** property.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Peek Overload List](#)

SocketWrench.Peek Method (Byte[], Int32)

Read data from the socket and store it in a byte array, but do not remove the data from the socket buffers.

[Visual Basic]

```
Overloads Public Function Peek( _  
    ByVal buffer As Byte(), _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Peek(  
    byte[] buffer,  
    int length  
);
```

Parameters

buffer

A byte array that the data will be stored in.

length

An integer value which specifies the maximum number of bytes of data to read. This value cannot be larger than the size of the buffer specified by the caller.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that there is no data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Peek** method returns data that is available to read from the socket, up to the number of bytes specified. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. Using this method in a loop to poll a non-blocking socket can cause the application to become non-responsive. To determine if there is data available to be read, use the **IsReadable** property.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Peek Overload List](#)

SocketWrench.Peek Method (String)

Read data from the socket and store it in a string, but do not remove the data from the socket buffers.

[Visual Basic]

```
Overloads Public Function Peek( _  
    ByRef buffer As String _  
) As Integer
```

[C#]

```
public int Peek(  
    ref string buffer  
);
```

Parameters

buffer

A string that will contain the data read from the socket.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that there is no data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Peek** method returns data that is available to read from the socket, up to a maximum of 8192 bytes. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. Using this method in a loop to poll a non-blocking socket can cause the application to become non-responsive. To determine if there is data available to be read, use the **IsReadable** property.

This method should only be used if the remote host is sending data that consists of printable characters. Binary data should be read using the method that accepts a byte array as the buffer parameter.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Peek Overload List](#)

SocketWrench.Peek Method (String, Int32)

Read data from the socket and store it in a string, but do not remove the data from the socket buffers.

[Visual Basic]

```
Overloads Public Function Peek( _  
    ByRef buffer As String, _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Peek(  
    ref string buffer,  
    int length  
);
```

Parameters

buffer

A string that will contain the data read from the socket.

length

An integer value which specifies the maximum number of bytes of data to read.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that there is no data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Peek** method returns data that is available to read from the socket, up to the number of bytes specified. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. Using this method in a loop to poll a non-blocking socket can cause the application to become non-responsive. To determine if there is data available to be read, use the **IsReadable** property.

This method should only be used if the remote host is sending data that consists of printable characters. Binary data should be read using the method that accepts a byte array as the buffer parameter.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Peek Overload List](#)

SocketWrench.Read Method

Read data from the socket and store it in a byte array.

Overload List

Read data from the socket and store it in a byte array.

```
public int Read(byte[]);
```

Read data from the socket and store it in a byte array.

```
public int Read(byte[],int);
```

Read data from the socket and store it in a string.

```
public int Read(ref string);
```

Read data from the socket and store it in a string.

```
public int Read(ref string,int);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Read Method (Byte[])

Read data from the socket and store it in a byte array.

[Visual Basic]

```
Overloads Public Function Read( _  
    ByVal buffer As Byte() _  
) As Integer
```

[C#]

```
public int Read(  
    byte[] buffer  
);
```

Parameters

buffer

A byte array that the data will be stored in.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Read** method returns data that has been read from the socket, up to the size of the byte array passed to the method. If no data is available to be read, an error will be generated if the socket is in non-blocking mode. If the socket is in blocking mode, the program will stop until data is received from the server or the connection is closed.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Read Overload List](#)

SocketWrench.Read Method (Byte[], Int32)

Read data from the socket and store it in a byte array.

[Visual Basic]

```
Overloads Public Function Read( _  
    ByVal buffer As Byte(), _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Read(  
    byte[] buffer,  
    int length  
);
```

Parameters

buffer

A byte array that the data will be stored in.

length

An integer value which specifies the maximum number of bytes of data to read. This value cannot be larger than the size of the buffer specified by the caller.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Read** method returns data that has been read from the socket, up to the number of bytes specified. If no data is available to be read, an error will be generated if the socket is in non-blocking mode. If the socket is in blocking mode, the program will stop until data is received from the server or the connection is closed.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Read Overload List](#)

SocketWrench.Read Method (String)

Read data from the socket and store it in a string.

[Visual Basic]

```
Overloads Public Function Read( _  
    ByRef buffer As String _  
) As Integer
```

[C#]

```
public int Read(  
    ref string buffer  
);
```

Parameters

buffer

A string that will contain the data read from the socket.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Read** method returns data that has been read from the socket, up to a maximum of 8192 bytes. If no data is available to be read, an error will be generated if the socket is in non-blocking mode. If the socket is in blocking mode, the program will stop until data is received from the server or the connection is closed.

This method should only be used if the remote host is sending data that consists of printable characters. Binary data should be read using the method that accepts a byte array as the buffer parameter.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Read Overload List](#)

SocketWrench.Read Method (String, Int32)

Read data from the socket and store it in a string.

[Visual Basic]

```
Overloads Public Function Read( _  
    ByRef buffer As String, _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Read(  
    ref string buffer,  
    int length  
);
```

Parameters

buffer

A string that will contain the data read from the socket.

length

An integer value which specifies the maximum number of bytes of data to read.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Read** method returns data that has been read from the socket, up to the number of bytes specified. If no data is available to be read, an error will be generated if the socket is in non-blocking mode. If the socket is in blocking mode, the program will stop until data is received from the server or the connection is closed.

This method should only be used if the remote host is sending data that consists of printable characters. Binary data should be read using the method that accepts a byte array as the buffer parameter.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Read Overload List](#)

SocketWrench.ReadFrom Method

Read data from the socket and store it in a byte array.

Overload List

Read data from the socket and store it in a byte array.

```
public int ReadFrom(byte[],int,ref string,ref int);
```

Read data from the socket and store it in a byte array.

```
public int ReadFrom(byte[],ref string,ref int);
```

Read data from the socket and store it in a string.

```
public int ReadFrom(ref string,int,ref string,ref int);
```

Read data from the socket and store it in a string.

```
public int ReadFrom(ref string,ref string,ref int);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.ReadFrom Method (Byte[], Int32, String, Int32)

Read data from the socket and store it in a byte array.

[Visual Basic]

```
Overloads Public Function ReadFrom( _  
    ByVal buffer As Byte(), _  
    ByVal length As Integer, _  
    ByRef hostAddress As String, _  
    ByRef hostPort As Integer _  
) As Integer
```

[C#]

```
public int ReadFrom(  
    byte[] buffer,  
    int length,  
    ref string hostAddress,  
    ref int hostPort  
);
```

Parameters

buffer

A byte array that the data will be stored in.

length

An integer value which specifies the maximum number of bytes of data to read. This value cannot be larger than the size of the buffer specified by the caller.

hostAddress

A string passed by reference that will contain the remote host Internet address when the method returns. For stream sockets, this will be the same as the address used to establish the connection. For datagram sockets, this will specify the address of host that sent the datagram.

hostPort

An integer passed by reference that will contain the remote host port number when the method returns. For stream sockets, this will be the same as the port number used to establish the connection. For datagram sockets, this will specify the port number used by the host that sent the datagram.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **ReadFrom** method returns data that has been read from the socket, up to the number of bytes specified. If no data is available to be read, an error will be generated if the socket is in non-blocking mode. If the socket is in blocking mode, the program will stop until data is received from the server or the connection is closed.

This method is typically used when reading binary data from a datagram socket.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ReadFrom Overload List](#)

SocketWrench.ReadFrom Method (Byte[], String, Int32)

Read data from the socket and store it in a byte array.

[Visual Basic]

```
Overloads Public Function ReadFrom( _  
    ByVal buffer As Byte(), _  
    ByRef hostAddress As String, _  
    ByRef hostPort As Integer _  
) As Integer
```

[C#]

```
public int ReadFrom(  
    byte[] buffer,  
    ref string hostAddress,  
    ref int hostPort  
);
```

Parameters

buffer

A byte array that the data will be stored in.

hostAddress

A string passed by reference that will contain the remote host Internet address when the method returns. For stream sockets, this will be the same as the address used to establish the connection. For datagram sockets, this will specify the address of host that sent the datagram.

hostPort

An integer passed by reference that will contain the remote host port number when the method returns. For stream sockets, this will be the same as the port number used to establish the connection. For datagram sockets, this will specify the port number used by the host that sent the datagram.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **ReadFrom** method returns data that has been read from the socket, up to the number of bytes specified. If no data is available to be read, an error will be generated if the socket is in non-blocking mode. If the socket is in blocking mode, the program will stop until data is received from the server or the connection is closed.

This method is typically used when reading binary data from a datagram socket.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ReadFrom Overload List](#)

SocketWrench.ReadFrom Method (String, Int32, String, Int32)

Read data from the socket and store it in a string.

[Visual Basic]

```
Overloads Public Function ReadFrom( _  
    ByRef buffer As String, _  
    ByVal length As Integer, _  
    ByRef hostAddress As String, _  
    ByRef hostPort As Integer _  
) As Integer
```

[C#]

```
public int ReadFrom(  
    ref string buffer,  
    int length,  
    ref string hostAddress,  
    ref int hostPort  
);
```

Parameters

buffer

A string that will contain the data read from the socket.

length

An integer value which specifies the maximum number of bytes of data to read.

hostAddress

A string passed by reference that will contain the remote host Internet address when the method returns. For stream sockets, this will be the same as the address used to establish the connection. For datagram sockets, this will specify the address of host that sent the datagram.

hostPort

An integer passed by reference that will contain the remote host port number when the method returns. For stream sockets, this will be the same as the port number used to establish the connection. For datagram sockets, this will specify the port number used by the host that sent the datagram.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **ReadFrom** method returns data that has been read from the socket, up to the number of bytes specified. If no data is available to be read, an error will be generated if the socket is in non-blocking mode. If the socket is in blocking mode, the program will stop until data is received from the server or the connection is closed.

This method should only be used if the remote host is sending data that consists of printable characters. Binary data should be read using the method that accepts a byte array as the buffer parameter.

This method is typically used when reading text data from a datagram socket.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ReadFrom Overload List](#)

SocketWrench.ReadFrom Method (String, String, Int32)

Read data from the socket and store it in a string.

[Visual Basic]

```
Overloads Public Function ReadFrom( _  
    ByRef buffer As String, _  
    ByRef hostAddress As String, _  
    ByRef hostPort As Integer _  
) As Integer
```

[C#]

```
public int ReadFrom(  
    ref string buffer,  
    ref string hostAddress,  
    ref int hostPort  
);
```

Parameters

buffer

A string that will contain the data read from the socket.

hostAddress

A string passed by reference that will contain the remote host Internet address when the method returns. For stream sockets, this will be the same as the address used to establish the connection. For datagram sockets, this will specify the address of host that sent the datagram.

hostPort

An integer passed by reference that will contain the remote host port number when the method returns. For stream sockets, this will be the same as the port number used to establish the connection. For datagram sockets, this will specify the port number used by the host that sent the datagram.

Return Value

An integer value which specifies the number of bytes actually read from the socket. A return value of zero specifies that the remote host has closed the connection and there is no more data available to be read. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **ReadFrom** method returns data that has been read from the socket, up to the maximum size of a datagram. If no data is available to be read, an error will be generated if the socket is in non-blocking mode. If the socket is in blocking mode, the program will stop until data is received from the server or the connection is closed.

This method should only be used if the remote host is sending data that consists of printable characters. Binary data should be read using the method that accepts a byte array as the buffer parameter.

This method is typically used when reading text data from a datagram socket.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ReadFrom Overload List](#)

SocketWrench.ReadLine Method

Read a line of data from the socket and return it in a string buffer.

Overload List

Read a line of data from the socket and return it in a string buffer.

```
public string ReadLine();
```

Read up to a line of data from the socket and return it in a string buffer.

```
public bool ReadLine(ref string);
```

Read up to a line of data from the socket and return it in a string buffer.

```
public bool ReadLine(ref string,int);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.ReadLine Method (String)

Read up to a line of data from the socket and return it in a string buffer.

[Visual Basic]

```
Overloads Public Function ReadLine( _  
    ByRef buffer As String _  
) As Boolean
```

[C#]

```
public bool ReadLine(  
    ref string buffer  
);
```

Parameters

buffer

A string which will contain the data read from the socket.

Return Value

This method returns a boolean value which specifies if a line of data has been read. A value of **true** indicates a line of data has been read. If an error occurs or there is no more data available to read, then the method will return **false**. It is possible for data to be returned in the string buffer even if the return value is **false**.

Applications should check the length of the string after the method returns to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Remarks

The **ReadLine** method reads data from the socket up to 8192 bytes in length or until an end-of-line character sequence is encountered. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a single line of text data in a string variable. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point; the string will not contain the carriage-return or linefeed characters.

There are some limitations when using the **ReadLine** method. The method should only be used to read text, never binary data. In particular, it will discard nulls, linefeed and carriage return control characters. This method will force the current thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection. If the **Blocking** property is set to **false**, calling this method will automatically switch the socket into a blocking mode, read the data and then restore the socket to non-blocking mode. If another socket operation is attempted while **ReadLine** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking socket connections.

The **Read** and **ReadLine** methods can be intermixed, however be aware that the **Read** method will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ReadLine Overload List](#)

SocketWrench.ReadLine Method (String, Int32)

Read up to a line of data from the socket and return it in a string buffer.

[Visual Basic]

```
Overloads Public Function ReadLine( _  
    ByRef buffer As String, _  
    ByVal length As Integer _  
) As Boolean
```

[C#]

```
public bool ReadLine(  
    ref string buffer,  
    int length  
);
```

Parameters

buffer

A string which will contain the data read from the socket.

length

An integer value which specifies the maximum number of bytes of data to read.

Return Value

This method returns a boolean value which specifies if a line of data has been read. A value of **true** indicates a line of data has been read. If an error occurs or there is no more data available to read, then the method will return **false**. It is possible for data to be returned in the string buffer even if the return value is **false**.

Applications should check the length of the string after the method returns to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Remarks

The **ReadLine** method reads data from the socket up to the specified number of bytes or until an end-of-line character sequence is encountered. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a single line of text data in a string variable. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point; the string will not contain the carriage-return or linefeed characters.

There are some limitations when using the **ReadLine** method. The method should only be used to read text, never binary data. In particular, it will discard nulls, linefeed and carriage return control characters. This method will force the current thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection. If the **Blocking** property is set to **false**, calling this method will automatically switch the socket into a blocking mode, read the data and then restore the socket to non-blocking mode. If another socket operation is attempted while **ReadLine** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking socket connections.

The **Read** and **ReadLine** methods can be intermixed, however be aware that the **Read** method will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

See Also

SocketWrench.ReadStream Method

Read a data stream from the socket and store it in the specified byte array.

Overload List

Read a data stream from the socket and store it in the specified byte array.

```
public bool ReadStream(byte[],ref int);
```

Read a data stream from the socket and store it in the specified byte array.

```
public bool ReadStream(byte[],ref int,byte[]);
```

Read a data stream from the socket and store it in the specified byte array.

```
public bool ReadStream(byte[],ref int,byte[],SocketStream);
```

Read a data stream from the socket and store it in the specified string.

```
public bool ReadStream(ref string);
```

Read a data stream from the socket and store it in the specified string.

```
public bool ReadStream(ref string,bool);
```

Read a data stream from the socket and store it in the specified string.

```
public bool ReadStream(ref string,ref int);
```

Read a data stream from the socket and store it in the specified string.

```
public bool ReadStream(ref string,ref int,bool);
```

Read a data stream from the socket and store it in the specified string.

```
public bool ReadStream(ref string,ref int,string,bool);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.ReadStream Method (Byte[], Int32)

Read a data stream from the socket and store it in the specified byte array.

[Visual Basic]

```
Overloads Public Function ReadStream( _  
    ByVal buffer As Byte(), _  
    ByRef length As Integer _  
) As Boolean
```

[C#]

```
public bool ReadStream(  
    byte[] buffer,  
    ref int length  
);
```

Parameters

buffer

A byte array that the data will be stored in.

length

An integer value passed by reference which specifies the maximum number of bytes of data to read. This value cannot be larger than the size of the buffer specified by the caller. When the method returns, this value will be updated with the actual number of bytes read from the socket.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

This method will force the current thread to block until the operation completes. If this method is called and the **Blocking** property is set to **false**, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **ReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for data to be returned in the buffer even if the method returns **false**. Applications should also check the value of the *length* parameter to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Because **ReadStream** can potentially cause the current thread to block for long periods of time as the data stream is being read, the class will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ReadStream Overload List](#)

SocketWrench.ReadStream Method (Byte[], Int32, Byte[])

Read a data stream from the socket and store it in the specified byte array.

[Visual Basic]

```
Overloads Public Function ReadStream( _  
    ByVal buffer As Byte(), _  
    ByRef length As Integer, _  
    ByVal marker As Byte() _  
) As Boolean
```

[C#]

```
public bool ReadStream(  
    byte[] buffer,  
    ref int length,  
    byte[] marker  
);
```

Parameters

buffer

A byte array that the data will be stored in.

length

An integer value passed by reference which specifies the maximum number of bytes of data to read. This value cannot be larger than the size of the buffer specified by the caller. When the method returns, this value will be updated with the actual number of bytes read from the socket.

marker

An array of bytes which is used to designate the logical end of the data stream. When this byte sequence is encountered by the method, it will stop reading and return to the caller. The buffer will contain all of the data read from the socket up to and including the end-of-stream marker.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

This method will force the current thread to block until the operation completes. If this method is called and the **Blocking** property is set to **false**, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **ReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for data to be returned in the buffer even if the method returns **false**. Applications should also check the value of the *length* parameter to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Because **ReadStream** can potentially cause the current thread to block for long periods of time as the data stream is being read, the class will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a

blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ReadStream Overload List](#)

SocketWrench.ReadStream Method (Byte[], Int32, Byte[], SocketStream)

Read a data stream from the socket and store it in the specified byte array.

[Visual Basic]

```
Overloads Public Function ReadStream( _  
    ByVal buffer As Byte(), _  
    ByRef length As Integer, _  
    ByVal marker As Byte(), _  
    ByVal options As SocketStream _  
) As Boolean
```

[C#]

```
public bool ReadStream(  
    byte[] buffer,  
    ref int length,  
    byte[] marker,  
    SocketStream options  
);
```

Parameters

buffer

A byte array that the data will be stored in.

length

An integer value passed by reference which specifies the maximum number of bytes of data to read. This value cannot be larger than the size of the buffer specified by the caller. When the method returns, this value will be updated with the actual number of bytes read from the socket.

marker

An array of bytes which is used to designate the logical end of the data stream. When this byte sequence is encountered by the method, it will stop reading and return to the caller. The buffer will contain all of the data read from the socket up to and including the end-of-stream marker.

options

One of the [SocketStream](#) enumeration values which specifies how the data is processed.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

This method will force the current thread to block until the operation completes. If this method is called and the **Blocking** property is set to **false**, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **ReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for data to be returned in the buffer even if the method returns **false**. Applications should also check the value of the *length* parameter to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the

application to process that data, regardless of the method return value.

Because **ReadStream** can potentially cause the current thread to block for long periods of time as the data stream is being read, the class will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ReadStream Overload List](#)

SocketWrench.ReadStream Method (String)

Read a data stream from the socket and store it in the specified string.

[Visual Basic]

```
Overloads Public Function ReadStream( _  
    ByRef buffer As String _  
) As Boolean
```

[C#]

```
public bool ReadStream(  
    ref string buffer  
);
```

Parameters

buffer

A string that will contain the data read from the socket.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

This method will force the current thread to block until the operation completes. If this method is called and the **Blocking** property is set to **false**, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **ReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for data to be returned in the buffer even if the method returns **false**. Applications should also check the length of the string to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Because **ReadStream** can potentially cause the current thread to block for long periods of time as the data stream is being read, the class will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ReadStream Overload List](#)

SocketWrench.ReadStream Method (String, Boolean)

Read a data stream from the socket and store it in the specified string.

[Visual Basic]

```
Overloads Public Function ReadStream( _  
    ByRef buffer As String, _  
    ByVal convertText As Boolean _  
) As Boolean
```

[C#]

```
public bool ReadStream(  
    ref string buffer,  
    bool convertText  
);
```

Parameters

buffer

A string that will contain the data read from the socket.

convertText

A boolean flag which specifies if the data data stream is considered to be textual and should be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

This method will force the current thread to block until the operation completes. If this method is called and the **Blocking** property is set to **false**, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **ReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for data to be returned in the buffer even if the method returns **false**. Applications should also check the length of the string to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Because **ReadStream** can potentially cause the current thread to block for long periods of time as the data stream is being read, the class will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ReadStream Overload List](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketWrench.ReadStream Method (String, Int32)

Read a data stream from the socket and store it in the specified string.

[Visual Basic]

```
Overloads Public Function ReadStream( _  
    ByRef buffer As String, _  
    ByRef length As Integer _  
) As Boolean
```

[C#]

```
public bool ReadStream(  
    ref string buffer,  
    ref int length  
);
```

Parameters

buffer

A string that will contain the data read from the socket.

length

An integer value passed by reference which specifies the maximum number of bytes of data to read. This value cannot be larger than the size of the buffer specified by the caller. When the method returns, this value will be updated with the actual number of bytes read from the socket.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

This method will force the current thread to block until the operation completes. If this method is called and the **Blocking** property is set to **false**, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **ReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for data to be returned in the buffer even if the method returns **false**. Applications should also check the value of the *length* parameter to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Because **ReadStream** can potentially cause the current thread to block for long periods of time as the data stream is being read, the class will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ReadStream Overload List](#)

SocketWrench.ReadStream Method (String, Int32, Boolean)

Read a data stream from the socket and store it in the specified string.

[Visual Basic]

```
Overloads Public Function ReadStream( _  
    ByRef buffer As String, _  
    ByRef length As Integer, _  
    ByVal convertText As Boolean _  
) As Boolean
```

[C#]

```
public bool ReadStream(  
    ref string buffer,  
    ref int length,  
    bool convertText  
);
```

Parameters

buffer

A string that will contain the data read from the socket.

length

An integer value passed by reference which specifies the maximum number of bytes of data to read. This value cannot be larger than the size of the buffer specified by the caller. When the method returns, this value will be updated with the actual number of bytes read from the socket.

convertText

A boolean flag which specifies if the data data stream is considered to be textual and should be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

This method will force the current thread to block until the operation completes. If this method is called and the **Blocking** property is set to **false**, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **ReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for data to be returned in the buffer even if the method returns **false**. Applications should also check the value of the *length* parameter to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Because **ReadStream** can potentially cause the current thread to block for long periods of time as the data stream is being read, the class will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ReadStream Overload List](#)

SocketWrench.ReadStream Method (String, Int32, String, Boolean)

Read a data stream from the socket and store it in the specified string.

[Visual Basic]

```
Overloads Public Function ReadStream( _  
    ByRef buffer As String, _  
    ByRef length As Integer, _  
    ByVal marker As String, _  
    ByVal convertText As Boolean _  
) As Boolean
```

[C#]

```
public bool ReadStream(  
    ref string buffer,  
    ref int length,  
    string marker,  
    bool convertText  
);
```

Parameters

buffer

A string that will contain the data read from the socket.

length

An integer value passed by reference which specifies the maximum number of bytes of data to read. This value cannot be larger than the size of the buffer specified by the caller. When the method returns, this value will be updated with the actual number of bytes read from the socket.

marker

A string which is used to designate the logical end of the data stream. When this character sequence is encountered by the method, it will stop reading and return to the caller. The string buffer will contain all of the data read from the socket up to and including the end-of-stream marker.

convertText

A boolean flag which specifies if the data data stream is considered to be textual and should be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

This method will force the current thread to block until the operation completes. If this method is called and the **Blocking** property is set to **false**, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **ReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to

manage each connection.

It is possible for data to be returned in the buffer even if the method returns **false**. Applications should also check the value of the *length* parameter to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Because **ReadStream** can potentially cause the current thread to block for long periods of time as the data stream is being read, the class will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.ReadStream Overload List](#)

SocketWrench.Reject Method

Rejects a connection request from a remote host.

[Visual Basic]

Public Function Reject() As Boolean

[C#]

public bool Reject();

Return Value

This method returns a boolean value. If the method succeeds, the return value is **true**. If the method fails, the return value is **false**. To get extended error information, check the value of the **LastError** property.

Remarks

The **Reject** method rejects a pending client connection and the remote host will see this as the connection being aborted. If there are no pending client connections at the time, this method will immediately return with an error indicating that the operation would cause the thread to block.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Reset Method

Reset the internal state of the object, resetting all properties to their default values.

[Visual Basic]

```
Public Sub Reset()
```

[C#]

```
public void Reset();
```

Remarks

The **Reset** method returns the object to its default state. If a socket has been allocated, it will be released and any active connections will be terminated. All properties will be reset to their default values.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Resolve Method

Resolves a host name to a host IP address.

[Visual Basic]

```
Public Function Resolve( _  
    ByVal hostName As String, _  
    ByRef hostAddress As String _  
) As Boolean
```

[C#]

```
public bool Resolve(  
    string hostName,  
    ref string hostAddress  
);
```

Parameters

hostName

A string which specifies the host name to be resolved.

hostAddress

A string which will contain the Internet address for the specified host.

Return Value

This method returns a boolean value. If the host name can be resolved, the return value is **true**. If the host name cannot be resolved, the return value is **false**. To get extended error information, check the value of the **LastError** property.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Shutdown Method

Disable sending data on the socket.

Overload List

Disable sending data on the socket.

```
public bool Shutdown();
```

Disable sending or receiving data on the socket.

```
public bool Shutdown(ShutdownOptions);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [Disconnect Method](#)

SocketWrench.Shutdown Method ()

Disable sending data on the socket.

[Visual Basic]

Overloads Public Function Shutdown() As Boolean

[C#]

public bool Shutdown();

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

In some asynchronous applications, it may be desirable for a client to inform the server that no further communication is wanted, while allowing the client to read any residual data that may reside in internal buffers on the client side. **Shutdown** accomplishes this because the socket handle is still valid after it has been called, although some or all communication with the remote host has ceased. Note that most applications do not typically need to use this method. To close a socket connection gracefully, you should use the **Disconnect** method.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Shutdown Overload List](#) | [Disconnect Method](#)

SocketWrench.Shutdown Method (ShutdownOptions)

Disable sending or receiving data on the socket.

[Visual Basic]

```
Overloads Public Function Shutdown( _  
    ByVal options As ShutdownOptions _  
) As Boolean
```

[C#]

```
public bool Shutdown(  
    ShutdownOptions options  
);
```

Parameters

options

One of the [ShutdownOptions](#) enumeration values which specifies the operation that will no longer be allowed.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

In some asynchronous applications, it may be desirable for a client to inform the server that no further communication is wanted, while allowing the client to read any residual data that may reside in internal buffers on the client side. **Shutdown** accomplishes this because the socket handle is still valid after it has been called, although some or all communication with the remote host has ceased.

Note that most applications do not typically need to use this method. To close a socket connection gracefully, you should use the **Disconnect** method.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Shutdown Overload List](#) | [Disconnect Method](#)

SocketWrench.StoreStream Method

Reads a data stream from the socket and stores it in the specified file.

Overload List

Reads a data stream from the socket and stores it in the specified file.

```
public bool StoreStream(string);
```

Reads a data stream from the socket and stores it in the specified file.

```
public bool StoreStream(string,ref int);
```

Reads a data stream from the socket and stores it in the specified file.

```
public bool StoreStream(string,ref int,bool);
```

Reads a data stream from the socket and stores it in the specified file.

```
public bool StoreStream(string,ref int,int,SocketStream);
```

Reads a data stream from the socket and stores it in the specified file.

```
public bool StoreStream(string,ref int,int,bool);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.StoreStream Method (String)

Reads a data stream from the socket and stores it in the specified file.

[Visual Basic]

```
Overloads Public Function StoreStream( _  
    ByVal fileName As String _  
) As Boolean
```

[C#]

```
public bool StoreStream(  
    string fileName  
);
```

Parameters

fileName

A string variable that specifies the name of the file that will contain the data read from the socket. If the file does not exist, it will be created. If the file does exist, the contents will be overwritten.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

This method will force the current thread to block until the operation completes. If this method is called with the **Blocking** property set to **false**, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **StoreStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

Because **StoreStream** can potentially cause the current thread to block for long periods of time as the data stream is being read, the class instance will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.StoreStream Overload List](#)

SocketWrench.StoreStream Method (String, Int32)

Reads a data stream from the socket and stores it in the specified file.

[Visual Basic]

```
Overloads Public Function StoreStream( _  
    ByVal fileName As String, _  
    ByRef length As Integer _  
) As Boolean
```

[C#]

```
public bool StoreStream(  
    string fileName,  
    ref int length  
);
```

Parameters

fileName

A string variable that specifies the name of the file that will contain the data read from the socket. If the file does not exist, it will be created. If the file does exist, the contents will be overwritten.

length

An integer value which specifies the maximum amount of data to be read from the socket. When the method returns, this variable will be updated with the actual number of bytes read. Note that because this argument is passed by reference and modified by the method, you must provide a variable, not a numeric constant. If the value is initialized to zero, this method will read data from the socket until the remote host disconnects or an error occurs.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

This method will force the current thread to block until the operation completes. If this method is called with the **Blocking** property set to **false**, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **StoreStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

Because **StoreStream** can potentially cause the current thread to block for long periods of time as the data stream is being read, the class instance will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.StoreStream Overload List](#)

SocketWrench.StoreStream Method (String, Int32, Boolean)

Reads a data stream from the socket and stores it in the specified file.

[Visual Basic]

```
Overloads Public Function StoreStream( _  
    ByVal fileName As String, _  
    ByRef length As Integer, _  
    ByVal convertText As Boolean _  
) As Boolean
```

[C#]

```
public bool StoreStream(  
    string fileName,  
    ref int length,  
    bool convertText  
);
```

Parameters

fileName

A string variable that specifies the name of the file that will contain the data read from the socket. If the file does not exist, it will be created. If the file does exist, the contents will be overwritten.

length

An integer value which specifies the maximum amount of data to be read from the socket. When the method returns, this variable will be updated with the actual number of bytes read. Note that because this argument is passed by reference and modified by the method, you must provide a variable, not a numeric constant. If the value is initialized to zero, this method will read data from the socket until the remote host disconnects or an error occurs.

convertText

A boolean flag which specifies if the data data stream is considered to be textual and should be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data stored in the file to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

This method will force the current thread to block until the operation completes. If this method is called with the **Blocking** property set to **false**, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **StoreStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

Because **StoreStream** can potentially cause the current thread to block for long periods of time as the data

stream is being read, the class instance will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.StoreStream Overload List](#)

SocketWrench.StoreStream Method (String, Int32, Int32, Boolean)

Reads a data stream from the socket and stores it in the specified file.

[Visual Basic]

```
Overloads Public Function StoreStream( _  
    ByVal fileName As String, _  
    ByRef length As Integer, _  
    ByVal offset As Integer, _  
    ByVal convertText As Boolean _  
) As Boolean
```

[C#]

```
public bool StoreStream(  
    string fileName,  
    ref int length,  
    int offset,  
    bool convertText  
);
```

Parameters

fileName

A string variable that specifies the name of the file that will contain the data read from the socket. If the file does not exist, it will be created. If the file does exist, the contents will be overwritten.

length

An integer value which specifies the maximum amount of data to be read from the socket. When the method returns, this variable will be updated with the actual number of bytes read. Note that because this argument is passed by reference and modified by the method, you must provide a variable, not a numeric constant. If the value is initialized to zero, this method will read data from the socket until the remote host disconnects or an error occurs.

offset

A numeric value which specifies the byte offset into the file where the method will start storing data read from the socket. Note that all data after this offset will be truncated. If a value of zero is specified, the file will be completely overwritten if it already exists.

convertText

A boolean flag which specifies if the data data stream is considered to be textual and should be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data stored in the file to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

This method will force the current thread to block until the operation completes. If this method is called with the **Blocking** property set to **false**, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is

attempted while **StoreStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

Because **StoreStream** can potentially cause the current thread to block for long periods of time as the data stream is being read, the class instance will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.StoreStream Overload List](#)

SocketWrench.StoreStream Method (String, Int32, Int32, SocketStream)

Reads a data stream from the socket and stores it in the specified file.

[Visual Basic]

```
Overloads Public Function StoreStream( _  
    ByVal fileName As String, _  
    ByRef length As Integer, _  
    ByVal offset As Integer, _  
    ByVal options As SocketStream _  
) As Boolean
```

[C#]

```
public bool StoreStream(  
    string fileName,  
    ref int length,  
    int offset,  
    SocketStream options  
);
```

Parameters

fileName

A string variable that specifies the name of the file that will contain the data read from the socket. If the file does not exist, it will be created. If the file does exist, the contents will be overwritten.

length

An integer value which specifies the maximum amount of data to be read from the socket. When the method returns, this variable will be updated with the actual number of bytes read. Note that because this argument is passed by reference and modified by the method, you must provide a variable, not a numeric constant. If the value is initialized to zero, this method will read data from the socket until the remote host disconnects or an error occurs.

offset

A numeric value which specifies the byte offset into the file where the method will start storing data read from the socket. Note that all data after this offset will be truncated. If a value of zero is specified, the file will be completely overwritten if it already exists.

options

One of the [SocketStream](#) enumeration values which specifies how the data is processed.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

This method will force the current thread to block until the operation completes. If this method is called with the **Blocking** property set to **false**, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **StoreStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

Because **StoreStream** can potentially cause the current thread to block for long periods of time as the data

stream is being read, the class will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.StoreStream Overload List](#)

SocketWrench.Uninitialize Method

Uninitialize the class library and release any resources allocated for the current thread.

[Visual Basic]

```
Public Sub Uninitialize()
```

[C#]

```
public void Uninitialize();
```

Remarks

The **Uninitialize** method terminates any active connection, releases resources allocated for the current thread and unloads the networking library. After this method has been called, no further socket operations may be performed until the class instance has been re-initialized.

If the **Initialize** method is explicitly called by the application, it should be matched by a call to the **Uninitialize** method when that instance of the class is no longer needed.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [Initialize Method](#)

SocketWrench.Write Method

Write one or more bytes of data to the socket.

Overload List

Write one or more bytes of data to the socket.

```
public int Write(byte[]);
```

Write one or more bytes of data to the socket.

```
public int Write(byte[],int);
```

Write a string of characters to the socket.

```
public int Write(string);
```

Write a string of characters to the socket.

```
public int Write(string,int);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.Write Method (Byte[])

Write one or more bytes of data to the socket.

[Visual Basic]

```
Overloads Public Function Write( _  
    ByVal buffer As Byte() _  
) As Integer
```

[C#]

```
public int Write(  
    byte[] buffer  
);
```

Parameters

buffer

A byte array that contains the data to be written to the socket.

Return Value

An integer value which specifies the number of bytes actually written to the socket. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Write** method sends one or more bytes of data to the remote host. If there is enough room in the socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space and the socket is in blocking mode, then the method will block until the data can be sent. If the socket is in non-blocking mode and the send buffer is full, an error will occur.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Write Overload List](#)

SocketWrench.Write Method (Byte[], Int32)

Write one or more bytes of data to the socket.

[Visual Basic]

```
Overloads Public Function Write( _  
    ByVal buffer As Byte(), _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Write(  
    byte[] buffer,  
    int length  
);
```

Parameters

buffer

A byte array that contains the data to be written to the socket.

length

An integer value which specifies the maximum number of bytes of data to write. This value cannot be larger than the size of the buffer specified by the caller.

Return Value

An integer value which specifies the number of bytes actually written to the socket. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Write** method sends one or more bytes of data to the remote host. If there is enough room in the socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space and the socket is in blocking mode, then the method will block until the data can be sent. If the socket is in non-blocking mode and the send buffer is full, an error will occur.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Write Overload List](#)

SocketWrench.Write Method (String)

Write a string of characters to the socket.

[Visual Basic]

```
Overloads Public Function Write( _  
    ByVal buffer As String _  
) As Integer
```

[C#]

```
public int Write(  
    string buffer  
);
```

Parameters

buffer

A string which contains the data to be written to the socket.

Return Value

An integer value which specifies the number of characters actually written to the socket. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Write** method sends a string of characters to the remote host. If there is enough room in the socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space and the socket is in blocking mode, then the method will block until the data can be sent. If the socket is in non-blocking mode and the send buffer is full, an error will occur.

The string will be converted to an array of bytes before being written to the socket. By default, the character encoding used will be for the current locale. Depending on the contents of the string, the number of bytes written may be different than the string length specified. This is because the conversion from Unicode to a byte array may result in a multi-byte character sequence.

You should never use strings to read and write binary data. Always use byte arrays to ensure that no character conversion is performed.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Write Overload List](#)

SocketWrench.Write Method (String, Int32)

Write a string of characters to the socket.

[Visual Basic]

```
Overloads Public Function Write( _  
    ByVal buffer As String, _  
    ByVal length As Integer _  
) As Integer
```

[C#]

```
public int Write(  
    string buffer,  
    int length  
);
```

Parameters

buffer

A string which contains the data to be written to the socket.

length

An integer value which specifies the maximum number of characters to write. This value cannot be larger than the length of the string specified by the caller.

Return Value

An integer value which specifies the number of characters actually written to the socket. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **Write** method sends a string of characters to the remote host. If there is enough room in the socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space and the socket is in blocking mode, then the method will block until the data can be sent. If the socket is in non-blocking mode and the send buffer is full, an error will occur.

The string will be converted to an array of bytes before being written to the socket. By default, the character encoding used will be for the current locale. Depending on the contents of the string, the number of bytes written may be different than the string length specified. This is because the conversion from Unicode to a byte array may result in a multi-byte character sequence.

You should never use strings to read and write binary data. Always use byte arrays to ensure that no character conversion is performed.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.Write Overload List](#)

SocketWrench.WriteLine Method

Send an empty line of text to the remote host, terminated by a carriage-return and linefeed.

Overload List

Send an empty line of text to the remote host, terminated by a carriage-return and linefeed.

```
public bool WriteLine();
```

Send a line of text to the remote host, terminated by a carriage-return and linefeed.

```
public bool WriteLine(string);
```

Send a line of text to the remote host, terminated by a carriage-return and linefeed.

```
public bool WriteLine(string, ref int);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.WriteLine Method ()

Send an empty line of text to the remote host, terminated by a carriage-return and linefeed.

[Visual Basic]

Overloads Public Function WriteLine() As Boolean

[C#]

public bool WriteLine();

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

The **WriteLine** method will send an empty line of text, terminated by a carriage-return and linefeed. Calling this method will force the application to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection. If this method is called with the **Blocking** property set to **false**, it will automatically switch the socket into a blocking mode, send the data and then restore the socket to non-blocking mode. If another socket operation is attempted while the **WriteLine** method is blocked sending data to the remote host, an error will occur. It is recommended that this method only be used with blocking socket connections.

The **Write** and **WriteLine** methods can be safely intermixed.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.WriteLine Overload List](#)

SocketWrench.WriteLine Method (String)

Send a line of text to the remote host, terminated by a carriage-return and linefeed.

[Visual Basic]

```
Overloads Public Function WriteLine( _  
    ByVal buffer As String _  
) As Boolean
```

[C#]

```
public bool WriteLine(  
    string buffer  
);
```

Parameters

buffer

A string which contains the data that will be sent to the remote host. The data will always be terminated with a carriage-return and linefeed control character sequence. If the string is empty, then a only a carriage-return and linefeed are written to the socket. Note that if the string contains a null character, any data that follows the null character will be discarded.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

The **WriteLine** method should only be used to send text, never binary data. In particular, this method will discard any data that follows a null character and will append linefeed and carriage return control characters to the data stream. Calling this method will force the current thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection. If this method is called with the **Blocking** property set to **false**, it will automatically switch the socket into a blocking mode, send the data and then restore the socket to non-blocking mode. If another socket operation is attempted while the **WriteLine** method is blocked sending data to the remote host, an error will occur. It is recommended that this method only be used with blocking socket connections.

The **Write** and **WriteLine** methods can be safely intermixed.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.WriteLine Overload List](#)

SocketWrench.WriteLine Method (String, Int32)

Send a line of text to the remote host, terminated by a carriage-return and linefeed.

[Visual Basic]

```
Overloads Public Function WriteLine( _  
    ByVal buffer As String, _  
    ByRef length As Integer _  
) As Boolean
```

[C#]

```
public bool WriteLine(  
    string buffer,  
    ref int length  
);
```

Parameters

buffer

A string which contains the data that will be sent to the remote host. The data will always be terminated with a carriage-return and linefeed control character sequence. If the string is empty, then a only a carriage-return and linefeed are written to the socket. Note that if the string contains a null character, any data that follows the null character will be discarded.

length

An integer value which specifies the maximum number of characters to write. This value cannot be larger than the length of the string specified by the caller.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

The **WriteLine** method should only be used to send text, never binary data. In particular, this method will discard any data that follows a null character and will append linefeed and carriage return control characters to the data stream. Calling this method will force the current thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection. If this method is called with the **Blocking** property set to **false**, it will automatically switch the socket into a blocking mode, send the data and then restore the socket to non-blocking mode. If another socket operation is attempted while the **WriteLine** method is blocked sending data to the remote host, an error will occur. It is recommended that this method only be used with blocking socket connections.

The **Write** and **WriteLine** methods can be safely intermixed.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.WriteLine Overload List](#)

SocketWrench.WriteStream Method

Write a stream of bytes to the socket.

Overload List

Write a stream of bytes to the socket.

```
public bool WriteStream(byte[],ref int);
```

Write a string of characters to the socket.

```
public bool WriteStream(string,ref int);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.WriteStream Method (Byte[], Int32)

Write a stream of bytes to the socket.

[Visual Basic]

```
Overloads Public Function WriteStream( _  
    ByVal buffer As Byte(), _  
    ByRef length As Integer _  
) As Boolean
```

[C#]

```
public bool WriteStream(  
    byte[] buffer,  
    ref int length  
);
```

Parameters

buffer

A byte array that contains the data to be written to the socket.

length

An integer value passed by reference which specifies the maximum number of bytes to write. This value cannot be larger than the size of the buffer specified by the caller. When the method returns, this value will be updated with the actual number of bytes written to the socket.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

The **WriteStream** method enables an application to write an arbitrarily large stream of data from a byte array to the socket. Unlike the **Write** method, which may not write all of the data in a single call, the **WriteStream** method will only return when all of the data has been written or an error occurs.

This method will force the current thread to block until the operation completes. If this method is called with the **Blocking** property set to **false**, it will automatically switch the socket into a blocking mode, write the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **WriteStream** is blocked sending data to the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible that some data will have been written to the socket even if the method returns **false**. Applications should also check the value of the *length* argument to determine if any data was sent. For example, if a timeout occurs while the function is waiting to write more data, it will return zero; however, some data may have already been written to the socket prior to the error condition.

Because **WriteStream** can potentially cause the current thread to block for long periods of time as the data stream is being written, the class instance will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being written. Note that an application should never perform a blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.WriteStream Overload List](#)

SocketWrench.WriteStream Method (String, Int32)

Write a string of characters to the socket.

[Visual Basic]

```
Overloads Public Function WriteStream( _  
    ByVal buffer As String, _  
    ByRef length As Integer _  
) As Boolean
```

[C#]

```
public bool WriteStream(  
    string buffer,  
    ref int length  
);
```

Parameters

buffer

A string that contains the data to be written to the socket.

length

An integer value passed by reference which specifies the maximum number of characters to write. This value cannot be larger than the length of the string specified by the caller. When the method returns, this value will be updated with the actual number of bytes written to the socket.

Return Value

A boolean value which specifies if the operation completed successfully. A return value of **false** indicates an error has occurred. To get extended error information, check the value of the **LastError** property.

Remarks

The **WriteStream** method enables an application to write an arbitrarily large stream of data from a string to the socket. Unlike the **Write** method, which may not write all of the data in a single call, the **WriteStream** method will only return when all of the data has been written or an error occurs.

This method will force the current thread to block until the operation completes. If this method is called with the **Blocking** property set to **false**, it will automatically switch the socket into a blocking mode, write the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **WriteStream** is blocked sending data to the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible that some data will have been written to the socket even if the method returns **false**. Applications should also check the value of the *length* argument to determine if any data was sent. For example, if a timeout occurs while the function is waiting to write more data, it will return zero; however, some data may have already been written to the socket prior to the error condition.

Because **WriteStream** can potentially cause the current thread to block for long periods of time as the data stream is being written, the class instance will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being written. Note that an application should never perform a blocking operation inside the event handler.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.WriteStream Overload List](#)

SocketWrench.WriteTo Method

Write one or more bytes of data to the socket.

Overload List

Write one or more bytes of data to the socket.

```
public int WriteTo(byte[],int,string,int);
```

Write one or more bytes of data to the socket.

```
public int WriteTo(byte[],string,int);
```

Write a string of characters to the socket.

```
public int WriteTo(string,int,string,int);
```

Write a string of characters to the socket.

```
public int WriteTo(string,string,int);
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.WriteTo Method (Byte[], Int32, String, Int32)

Write one or more bytes of data to the socket.

[Visual Basic]

```
Overloads Public Function WriteTo( _  
    ByVal buffer As Byte(), _  
    ByVal length As Integer, _  
    ByVal hostAddress As String, _  
    ByVal hostPort As Integer _  
) As Integer
```

[C#]

```
public int WriteTo(  
    byte[] buffer,  
    int length,  
    string hostAddress,  
    int hostPort  
);
```

Parameters

buffer

A byte array that contains the data to be written to the socket.

length

An integer value which specifies the maximum number of bytes of data to write. This value cannot be larger than the size of the buffer specified by the caller.

hostAddress

A string value which specifies the address of the remote host that the data will be sent to. For datagram sockets, this may be any valid Internet address. For stream sockets, this must be the same address that was used to establish the connection.

hostPort

An integer value which specifies the port number of the remote host that the data will be sent to. For datagram sockets, this may be any valid port number. For stream sockets, this must be the same port number that was used to establish the connection.

Return Value

An integer value which specifies the number of bytes actually written to the socket. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **WriteTo** method sends one or more bytes of data to the remote host. If there is enough room in the socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space and the socket is in blocking mode, then the method will block until the data can be sent. If the socket is in non-blocking mode and the send buffer is full, an error will occur.

This method is typically used when writing binary data to a datagram socket.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.WriteTo Overload List](#)

SocketWrench.WriteTo Method (Byte[], String, Int32)

Write one or more bytes of data to the socket.

[Visual Basic]

```
Overloads Public Function WriteTo( _  
    ByVal buffer As Byte(), _  
    ByVal hostAddress As String, _  
    ByVal hostPort As Integer _  
) As Integer
```

[C#]

```
public int WriteTo(  
    byte[] buffer,  
    string hostAddress,  
    int hostPort  
);
```

Parameters

buffer

A byte array that contains the data to be written to the socket.

hostAddress

A string value which specifies the address of the remote host that the data will be sent to. For datagram sockets, this may be any valid Internet address. For stream sockets, this must be the same address that was used to establish the connection.

hostPort

An integer value which specifies the port number of the remote host that the data will be sent to. For datagram sockets, this may be any valid port number. For stream sockets, this must be the same port number that was used to establish the connection.

Return Value

An integer value which specifies the number of bytes actually written to the socket. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **WriteTo** method sends one or more bytes of data to the remote host. If there is enough room in the socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space and the socket is in blocking mode, then the method will block until the data can be sent. If the socket is in non-blocking mode and the send buffer is full, an error will occur.

This method is typically used when writing binary data to a datagram socket.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.WriteTo Overload List](#)

SocketWrench.WriteTo Method (String, Int32, String, Int32)

Write a string of characters to the socket.

[Visual Basic]

```
Overloads Public Function WriteTo( _  
    ByVal buffer As String, _  
    ByVal length As Integer, _  
    ByVal hostAddress As String, _  
    ByVal hostPort As Integer _  
) As Integer
```

[C#]

```
public int WriteTo(  
    string buffer,  
    int length,  
    string hostAddress,  
    int hostPort  
);
```

Parameters

buffer

A string that contains the data to be written to the socket.

length

An integer value which specifies the maximum number of characters to write. This value cannot be larger than the length of the string specified by the caller.

hostAddress

A string value which specifies the address of the remote host that the data will be sent to. For datagram sockets, this may be any valid Internet address. For stream sockets, this must be the same address that was used to establish the connection.

hostPort

An integer value which specifies the port number of the remote host that the data will be sent to. For datagram sockets, this may be any valid port number. For stream sockets, this must be the same port number that was used to establish the connection.

Return Value

An integer value which specifies the number of bytes actually written to the socket. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **WriteTo** method sends a string of characters to the remote host. If there is enough room in the socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space and the socket is in blocking mode, then the method will block until the data can be sent. If the socket is in non-blocking mode and the send buffer is full, an error will occur.

This method is typically used when writing text data to a datagram socket.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.WriteTo Overload List](#)

SocketWrench.WriteTo Method (String, String, Int32)

Write a string of characters to the socket.

[Visual Basic]

```
Overloads Public Function WriteTo( _  
    ByVal buffer As String, _  
    ByVal hostAddress As String, _  
    ByVal hostPort As Integer _  
) As Integer
```

[C#]

```
public int WriteTo(  
    string buffer,  
    string hostAddress,  
    int hostPort  
);
```

Parameters

buffer

A string that contains the data to be written to the socket.

hostAddress

A string value which specifies the address of the remote host that the data will be sent to. For datagram sockets, this may be any valid Internet address. For stream sockets, this must be the same address that was used to establish the connection.

hostPort

An integer value which specifies the port number of the remote host that the data will be sent to. For datagram sockets, this may be any valid port number. For stream sockets, this must be the same port number that was used to establish the connection.

Return Value

An integer value which specifies the number of bytes actually written to the socket. If an error occurs, a value of -1 is returned and the application should check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **WriteTo** method sends a string of characters to the remote host. If there is enough room in the socket's internal send buffer to accommodate all of the data, it is copied to the send buffer and control immediately returns to the caller. If amount of data exceeds the available buffer space and the socket is in blocking mode, then the method will block until the data can be sent. If the socket is in non-blocking mode and the send buffer is full, an error will occur.

This method is typically used when writing text data to a datagram socket.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#) | [SocketWrench.WriteTo Overload List](#)

SocketWrench Events

The events of the **SocketWrench** class are listed below. For a complete list of **SocketWrench** class members, see the [SocketWrench Members](#) topic.

Public Instance Events

OnAccept	Occurs when a remote host attempts to establish a connection with the local system.
OnCancel	Occurs when a blocking socket operation is canceled.
OnConnect	Occurs when a connection is established with the remote host.
OnDisconnect	Occurs when the remote host disconnects from the local system.
OnError	Occurs when an socket operation fails.
OnProgress	Occurs as a data stream is being read or written to the socket.
OnRead	Occurs when data is available to be read from the socket.
OnTimeout	Occurs when a blocking operation fails to complete before the timeout period elapses.
OnWrite	Occurs when data can be written to the socket.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.OnAccept Event

Occurs when a remote host attempts to establish a connection with the local system.

[Visual Basic]

Public Event OnAccept As OnAcceptEventHandler

[C#]

public event OnAcceptEventHandler OnAccept;

Event Data

The event handler receives an argument of type [SocketWrench.AcceptEventArgs](#) containing data related to this event. The following **SocketWrench.AcceptEventArgs** property provides information specific to this event.

Property	Description
Handle	Gets a value that specifies the socket handle for the listening server.

Remarks

The **OnAccept** event occurs when a remote host attempts to connect to the local system. A connection is not actually established until it has been accepted by the listening server. To accept the connection, the application must call the **Accept** method.

The **PeerAddress** or **PeerName** properties may be used to determine the Internet address and host name of the remote host that is establishing the connection. Note that this information may not be available until after the **Accept** method is called to accept the connection.

This event is only generated if the socket is in non-blocking mode.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.AcceptEventArgs Class

Provides data for the OnAccept event.

For a list of all members of this type, see [SocketWrench.AcceptEventArgs Members](#).

System.Object

System.EventArgs

SocketTools.SocketWrench.AcceptEventArgs

[Visual Basic]

Public Class SocketWrench.AcceptEventArgs

Inherits EventArgs

[C#]

public class SocketWrench.AcceptEventArgs : EventArgs

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

AcceptEventArgs specifies the socket handle for the server that should accept the incoming client connection.

The [OnAccept](#) event occurs when a remote host attempts to establish a connection with the local system.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketWrench.AcceptEventArgs Members](#) | [SocketTools Namespace](#)

SocketWrench.AcceptEventArgs Members

[SocketWrench.AcceptEventArgs overview](#)

Public Instance Constructors

 SocketWrench.AcceptEventArgs Constructor	Initializes a new instance of the SocketWrench.AcceptEventArgs class.

Public Instance Properties

Handle	Gets a value that specifies the socket handle for the listening server.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[SocketWrench.AcceptEventArgs Class](#) | [SocketTools Namespace](#)

SocketWrench.AcceptEventArgs Constructor

Initializes a new instance of the [SocketWrench.AcceptEventArgs](#) class.

[Visual Basic]

```
Public Sub New()
```

[C#]

```
public SocketWrench.AcceptEventArgs();
```

See Also

[SocketWrench.AcceptEventArgs Class](#) | [SocketTools Namespace](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketWrench.AcceptEventArgs Properties

The properties of the **SocketWrench.AcceptEventArgs** class are listed below. For a complete list of **SocketWrench.AcceptEventArgs** class members, see the [SocketWrench.AcceptEventArgs Members](#) topic.

Public Instance Properties

Handle	Gets a value that specifies the socket handle for the listening server.

See Also

[SocketWrench.AcceptEventArgs Class](#) | [SocketTools Namespace](#)

SocketWrench.AcceptEventArgs.Handle Property

Gets a value that specifies the socket handle for the listening server.

[Visual Basic]

Public ReadOnly Property Handle As Integer

[C#]

public int Handle {get;}

Property Value

An integer value which specifies the server socket handle.

See Also

[SocketWrench.AcceptEventArgs Class](#) | [SocketTools Namespace](#)

SocketWrench.OnCancel Event

Occurs when a blocking socket operation is canceled.

[Visual Basic]

```
Public Event OnCancel As EventHandler
```

[C#]

```
public event EventHandler OnCancel;
```

Remarks

The **OnCancel** event is generated when a blocking socket operation, such as sending or receiving data, is canceled with the **Cancel** method. To assist in determining which operation was canceled, check the value of the **Status** property.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.OnConnect Event

Occurs when a connection is established with the remote host.

[Visual Basic]

```
Public Event OnConnect As EventHandler
```

[C#]

```
public event EventHandler OnConnect;
```

Remarks

The **OnConnect** event occurs when a connection is made with a remote host as a result of a **Connect** method call. When the **Connect** method is called and the **Blocking** property is set to **false**, a socket is created but the connection is not actually established until after this event occurs. Between the time connection process is started and this event fires, no operation may be performed on the socket other than calling the **Disconnect** method.

This event is only generated if the socket is in non-blocking mode.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.OnDisconnect Event

Occurs when the remote host disconnects from the local system.

[Visual Basic]

```
Public Event OnDisconnect As EventHandler
```

[C#]

```
public event EventHandler OnDisconnect;
```

Remarks

The **OnDisconnect** event occurs when the remote host closes its socket, terminating its connection with the application. Because there may still be data in the socket receive buffers, you should continue to read data from the socket until the **Read** method returns a value of 0. Once all of the data has been read, you should call the **Disconnect** method to close the local socket and terminate the session.

This event is only generated if the socket is in non-blocking mode.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.OnError Event

Occurs when an socket operation fails.

```
[Visual Basic]
Public Event OnError As OnErrorEventHandler
```

```
[C#]
public event OnErrorEventHandler OnError;
```

Event Data

The event handler receives an argument of type [SocketWrench.ErrorEventArgs](#) containing data related to this event. The following **SocketWrench.ErrorEventArgs** properties provide information specific to this event.

Property	Description
Description	Gets a value which describes the last error that has occurred.
Error	Gets a value which specifies the last error that has occurred.

Remarks

The **OnError** event occurs when a socket operation fails.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.ErrorEventArgs Class

Provides data for the OnError event.

For a list of all members of this type, see [SocketWrench.ErrorEventArgs Members](#).

System.Object

System.EventArgs

SocketTools.SocketWrench.ErrorEventArgs

[Visual Basic]

Public Class SocketWrench.ErrorEventArgs

Inherits EventArgs

[C#]

public class SocketWrench.ErrorEventArgs : EventArgs

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

ErrorEventArgs specifies the numeric error code and a description of the error that has occurred.

An [OnError](#) event occurs when a method fails.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketWrench.ErrorEventArgs Members](#) | [SocketTools Namespace](#)

SocketWrench.ErrorEventArgs Members

[SocketWrench.ErrorEventArgs overview](#)

Public Instance Constructors

 SocketWrench.ErrorEventArgs Constructor	Initializes a new instance of the SocketWrench.ErrorEventArgs class.

Public Instance Properties

Description	Gets a value which describes the last error that has occurred.
Error	Gets a value which specifies the last error that has occurred.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[SocketWrench.ErrorEventArgs Class](#) | [SocketTools Namespace](#)

SocketWrench.ErrorEventArgs Constructor

Initializes a new instance of the [SocketWrench.ErrorEventArgs](#) class.

[Visual Basic]

```
Public Sub New()
```

[C#]

```
public SocketWrench.ErrorEventArgs();
```

See Also

[SocketWrench.ErrorEventArgs Class](#) | [SocketTools Namespace](#)

SocketWrench.ErrorEventArgs Properties

The properties of the **SocketWrench.ErrorEventArgs** class are listed below. For a complete list of **SocketWrench.ErrorEventArgs** class members, see the [SocketWrench.ErrorEventArgs Members](#) topic.

Public Instance Properties

Description	Gets a value which describes the last error that has occurred.
Error	Gets a value which specifies the last error that has occurred.

See Also

[SocketWrench.ErrorEventArgs Class](#) | [SocketTools Namespace](#)

SocketWrench.ErrorEventArgs.Description Property

Gets a value which describes the last error that has occurred.

[Visual Basic]

Public ReadOnly Property Description As String

[C#]

public string Description {get;}

Property Value

A string which describes the last error that has occurred.

See Also

[SocketWrench.ErrorEventArgs Class](#) | [SocketTools Namespace](#) | [Error Property](#)

SocketWrench.ErrorEventArgs.Error Property

Gets a value which specifies the last error that has occurred.

[Visual Basic]

Public **ReadOnly** **Property** **Error** As [ErrorCode](#)

[C#]

public [SocketWrench.ErrorCode](#) **Error** {**get**;}

Property Value

[ErrorCode](#) enumeration which specifies the error.

See Also

[SocketWrench.ErrorEventArgs Class](#) | [SocketTools Namespace](#) | [Description Property](#)

SocketWrench.OnProgress Event

Occurs as a data stream is being read or written to the socket.

[Visual Basic]

Public Event OnProgress As [OnProgressEventHandler](#)

[C#]

public event [OnProgressEventHandler](#) OnProgress;

Event Data

The event handler receives an argument of type [SocketWrench.ProgressEventArgs](#) containing data related to this event. The following **SocketWrench.ProgressEventArgs** properties provide information specific to this event.

Property	Description
BytesCopied	Gets a value which specifies the number of bytes of data that has been read or written.
BytesTotal	Gets a value which specifies the total number of bytes in the data stream.
Percent	Gets a value which specifies the percentage of data that has been read or written.

Remarks

The **OnProgress** event occurs as a data stream is being read or written to the socket. If large amounts of data are being read or written, this event can be used to update a progress bar or other user-interface component to provide the user with some visual feedback on the progress of the operation.

This event is only generated when the **ReadStream**, **WriteStream** or **StoreStream** methods are called.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.ProgressEventArgs Class

Provides data for the OnProgress event.

For a list of all members of this type, see [SocketWrench.ProgressEventArgs Members](#).

System.Object

System.EventArgs

SocketTools.SocketWrench.ProgressEventArgs

[Visual Basic]

Public Class SocketWrench.ProgressEventArgs

Inherits EventArgs

[C#]

public class SocketWrench.ProgressEventArgs : EventArgs

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

ProgressEventArgs specifies the number of bytes copied from the data stream, the total number of bytes in the data stream and a completion percentage.

The [OnProgress](#) event occurs as a data stream is being read or written to the socket. This event only occurs when the **ReadStream**, **WriteStream** or **StoreStream** methods are called.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketWrench.ProgressEventArgs Members](#) | [SocketTools Namespace](#)

SocketWrench.ProgressEventArgs Members

[SocketWrench.ProgressEventArgs overview](#)

Public Instance Constructors

 SocketWrench.ProgressEventArgs Constructor	Initializes a new instance of the SocketWrench.ProgressEventArgs class.

Public Instance Properties

BytesCopied	Gets a value which specifies the number of bytes of data that has been read or written.
BytesTotal	Gets a value which specifies the total number of bytes in the data stream.
Percent	Gets a value which specifies the percentage of data that has been read or written.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object .
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString (inherited from Object)	Returns a String that represents the current Object .

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object .

See Also

[SocketWrench.ProgressEventArgs Class](#) | [SocketTools Namespace](#)

SocketWrench.ProgressEventArgs Constructor

Initializes a new instance of the [SocketWrench.ProgressEventArgs](#) class.

[Visual Basic]

```
Public Sub New()
```

[C#]

```
public SocketWrench.ProgressEventArgs();
```

See Also

[SocketWrench.ProgressEventArgs Class](#) | [SocketTools Namespace](#)

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

SocketWrench.ProgressEventArgs Properties

The properties of the **SocketWrench.ProgressEventArgs** class are listed below. For a complete list of **SocketWrench.ProgressEventArgs** class members, see the [SocketWrench.ProgressEventArgs Members](#) topic.

Public Instance Properties

BytesCopied	Gets a value which specifies the number of bytes of data that has been read or written.
BytesTotal	Gets a value which specifies the total number of bytes in the data stream.
Percent	Gets a value which specifies the percentage of data that has been read or written.

See Also

[SocketWrench.ProgressEventArgs Class](#) | [SocketTools Namespace](#)

SocketWrench.ProgressEventArgs.BytesCopied Property

Gets a value which specifies the number of bytes of data that has been read or written.

[Visual Basic]

Public ReadOnly Property BytesCopied As Integer

[C#]

public int BytesCopied {get;}

Property Value

An integer value which specifies the number of bytes of data.

Remarks

The **BytesCopied** property specifies the number of bytes that have been read from the socket and stored in the local stream buffer, or written from the stream buffer to the socket.

See Also

[SocketWrench.ProgressEventArgs Class](#) | [SocketTools Namespace](#) | [BytesTotal Property](#) | [Percent Property](#)

SocketWrench.ProgressEventArgs.BytesTotal Property

Gets a value which specifies the total number of bytes in the data stream.

[Visual Basic]

```
Public ReadOnly Property BytesTotal As Integer
```

[C#]

```
public int BytesTotal {get;}
```

Property Value

An integer value which specifies the number of bytes of data.

Remarks

The **BytesTotal** property specifies the total amount of data being read from the socket and stored in the data stream, or written from the data stream to the socket. If the amount of data was unknown or unspecified at the time the method call was made, then this value will always be the same as the **BytesCopied** property.

See Also

[SocketWrench.ProgressEventArgs Class](#) | [SocketTools Namespace](#) | [BytesCopied Property](#) | [Percent Property](#)

SocketWrench.ProgressEventArgs.Percent Property

Gets a value which specifies the percentage of data that has been read or written.

[Visual Basic]

```
Public ReadOnly Property Percent As Integer
```

[C#]

```
public int Percent {get;}
```

Property Value

An integer value which specifies a percentage.

Remarks

The **Percent** property specifies the percentage of data that has been transmitted, expressed as an integer value between 0 and 100, inclusive. If the maximum size of the data stream was not specified by the caller, this value will always be 100.

See Also

[SocketWrench.ProgressEventArgs Class](#) | [SocketTools Namespace](#) | [BytesCopied Property](#) | [BytesTotal Property](#)

SocketWrench.OnRead Event

Occurs when data is available to be read from the socket.

[Visual Basic]

```
Public Event OnRead As EventHandler
```

[C#]

```
public event EventHandler OnRead;
```

Remarks

The **OnRead** event occurs when data is available to be read from the socket. This event is level-triggered, which means that once this event fires, it will not occur again until some data has been read from the socket. This design prevents an application from being flooded with event notifications. It is recommended that your application read all of the available data from the socket and store it in a local buffer for processing. See the example below.

This event is only generated if the socket is in non-blocking mode.

Example

```
Private Sub Socket_OnRead(ByVal sender As Object, ByVal e As System.EventArgs) Handles
Socket.OnRead
    Dim strBuffer As String
    Dim nRead As Integer

    Do
        ' Read up to m_nBufferSize bytes of data from the socket
        nRead = Socket.Read(strBuffer, m_nBufferSize)

        If nRead > 0 Then
            ' Append the data to an internal buffer for processing
            m_dataBuffer = m_dataBuffer + strBuffer
        End If
    Loop Until nRead < 1

    ProcessData()
End Sub
```

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.OnTimeout Event

Occurs when a blocking operation fails to complete before the timeout period elapses.

[Visual Basic]

Public Event OnTimeout As EventHandler

[C#]

public event EventHandler OnTimeout;

Remarks

The **OnTimeout** event occurs when a blocking operation, such as sending or receiving data on the socket, fails to complete before the specified timeout period elapses. The timeout period for a blocking operation can be adjusted by setting the **Timeout** property.

This event is only generated if the socket is in blocking mode.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.OnWrite Event

Occurs when data can be written to the socket.

[Visual Basic]

```
Public Event OnWrite As EventHandler
```

[C#]

```
public event EventHandler OnWrite;
```

Remarks

The **OnWrite** event occurs when the application can write data to the socket. This event will typically occur when a connection is first established with the remote host, and after the **Write** method has failed because there was insufficient memory available in the socket send buffers. In the second case, when some of the buffered data has been successfully sent to the remote host and there is space available in the send buffers, this event is used to signal the application that it may attempt to send more data.

This event is only generated if the socket is in non-blocking mode.

See Also

[SocketWrench Class](#) | [SocketTools Namespace](#)

SocketWrench.OnAcceptEventHandler Delegate

Represents the method that will handle the [OnAccept](#) event.

[Visual Basic]

```
Public Delegate Sub SocketWrench.OnAcceptEventHandler( _  
    ByVal sender As Object, _  
    ByVal e As AcceptEventArgs _  
)
```

[C#]

```
public delegate void SocketWrench.OnAcceptEventHandler(  
    object sender,  
    AcceptEventArgs e  
);
```

Parameters

sender

The source of the event.

e

An [AcceptEventArgs](#) that contains the event data.

Remarks

When you create an **OnAcceptEventHandler** delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, until you remove the delegate.

Note that the declaration of your event handler must have the same parameters as the **OnAcceptEventHandler** delegate declaration.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#) | [Accept Method](#) | [OnAccept Event](#)

SocketWrench.OnErrorHandler Delegate

Represents the method that will handle the [OnError](#) event.

[Visual Basic]

```
Public Delegate Sub SocketWrench.OnErrorHandler( _  
    ByVal sender As Object, _  
    ByVal e As EventArgs _  
)
```

[C#]

```
public delegate void SocketWrench.OnErrorHandler(  
    object sender,  
    EventArgs e  
);
```

Parameters

sender

The source of the event.

e

An [EventArgs](#) that contains the event data.

Remarks

When you create an **OnErrorEventHandler** delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, until you remove the delegate.

Note that the declaration of your event handler must have the same parameters as the **OnErrorEventHandler** delegate declaration.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#)

SocketWrench.OnProgressEventHandler Delegate

Represents the method that will handle the [OnProgress](#) event.

[Visual Basic]

```
Public Delegate Sub SocketWrench.OnProgressEventHandler( _  
    ByVal sender As Object, _  
    ByVal e As ProgressEventArgs _  
)
```

[C#]

```
public delegate void SocketWrench.OnProgressEventHandler(  
    object sender,  
    ProgressEventArgs e  
);
```

Parameters

sender

The source of the event.

e

A [ProgressEventArgs](#) that contains the event data.

Remarks

When you create an **OnProgressEventHandler** delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, until you remove the delegate.

Note that the declaration of your event handler must have the same parameters as the **OnProgressEventHandler** delegate declaration.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#)

SocketWrench.AddressType Enumeration

The supported Internet Protocol (IP) versions.

```
[Visual Basic]
Public Enum SocketWrench.AddressType
```

```
[C#]
public enum SocketWrench.AddressType
```

Remarks

The SocketWrench class uses the **AddressType** enumeration to specify which version of the Internet Protocol will be used when a socket is created. The default protocol used by the class when establishing a connection is **addressIpv4**.

Members

Member Name	Description
Unknown	The version of the Internet Protocol is not specified. When a connection is established, the most appropriate version of the protocol will be used with preference given to IPv4 connections whenever possible.
Ipv4	Internet Protocol version 4 will be used when establishing connections. It is recommended that most applications use this version of the protocol. IPv4 addresses are 32 bits in length and are specified as four integer values separated by periods. An example of an IPv4 address is 192.168.0.121
Ipv6	Internet Protocol version 6 will be used when establishing connections. IPv6 addresses are 128 bits in length and are specified as a series of hexadecimal values separated by colons. An example of an IPv6 address is fd7c:2f6a:4f4f:ba34::a32

Requirements

- Namespace:** [SocketTools](#)
- Assembly:** SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#) | [AddressFamily Property \(SocketTools.SocketWrench\)](#)

SocketWrench.ErrorCode Enumeration

The error codes used by the class.

[Visual Basic]

```
Public Enum SocketWrench.ErrorCode
```

[C#]

```
public enum SocketWrench.ErrorCode
```

Remarks

The **ErrorCode** enumeration specifies what error has occurred when a method fails. The current error code may be determined by checking the value of the **LastError** property.

Note that the last error code is only meaningful if the previous operation has failed.

Members

Member Name	Description
None	No error.
NotHandleOwner	Handle not owned by the current thread.
FileNotFound	The specified file or directory does not exist.
FileNotCreated	The specified file could not be created.
OperationCanceled	The blocking operation has been canceled.
InvalidFileType	The specified file is a block or character device, not a regular file.
InvalidDevice	The specified file type is invalid or not a regular file.
TooManyParameters	The maximum number of function parameters has been exceeded.
InvalidFileName	The specified file name contains invalid characters or is too long.
InvalidFileHandle	Invalid file handle passed to function.
FileReadFailed	Unable to read data from the specified file.
FileWriteFailed	Unable to write data to the specified file.
OutOfMemory	Out of memory.
AccessDenied	Access denied.
InvalidParameter	Invalid argument passed to function.
ClipboardUnavailable	The system clipboard is currently unavailable.
ClipboardEmpty	The system clipboard is empty or does not contain any text data.
FileEmpty	The specified file does not contain any data.
FileExists	The specified file already exists.
EndOfFile	End of file.

DeviceNotFound	The specified device could not be found.
DirectoryNotFound	The specified directory could not be found.
InvalidBuffer	Invalid memory address passed to function.
BufferTooSmall	The specified buffer is not large enough to contain the data.
NoHandles	No more handles are available to this process.
OperationWouldBlock	The specified operation would block the current thread.
OperationInProgress	A blocking operation is currently in progress.
AlreadyInProgress	The specified operation is already in progress.
InvalidHandle	Invalid handle passed to function.
InvalidAddress	Invalid network address specified.
InvalidSize	Datagram is too large to fit in specified buffer.
InvalidProtocol	Invalid network protocol specified.
ProtocolNotAvailable	The specified network protocol is not available.
ProtocolNotSupported	The specified protocol is not supported.
SocketNotSupported	The specified socket type is not supported.
InvalidOption	The specified option is invalid.
ProtocolFamily	Specified protocol family is not supported.
ProtocolAddress	The specified address is invalid for this protocol family.
AddressInUse	The specified address is in use by another process.
AddressUnavailable	The specified address cannot be assigned.
NetworkUnavailable	The networking subsystem is unavailable.
NetworkUnreachable	The specified network is unreachable.
NetworkReset	Network dropped connection on remote reset.
ConnectionAborted	Connection was aborted due to timeout or other failure.
ConnectionReset	Connection was reset by remote network.
OutOfBuffers	No buffer space is available.
AlreadyConnected	Connection already established with remote host.
NotConnected	No connection established with remote host.
ConnectionShutdown	Unable to send or receive data after connection shutdown.
OperationTimeout	The specified operation has timed out.
ConnectionRefused	The connection has been refused by the remote host.
HostUnavailable	The specified host is unavailable.

HostUnreachable	Remote host is unreachable.
TooManyProcesses	Too many processes are using the networking subsystem.
TooManyThreads	Too many threads have been created by the current process.
TooManySessions	Too many client sessions have been created by the current process.
InternalFailure	An unexpected internal error has occurred.
NetworkNotReady	Network subsystem is not ready for communication.
InvalidVersion	This version of the operating system is not supported.
NetworkNotInitialized	The networking subsystem has not been initialized.
RemoteShutdown	The remote host has initiated a graceful shutdown sequence.
InvalidHostName	The specified hostname is invalid or could not be resolved.
HostNameNotFound	The specified hostname could not be found.
HostNameRefused	Unable to resolve hostname, request refused.
HostNameNotResolved	Unable to resolve hostname, no address for specified host.
InvalidLicense	The license for this product is invalid.
ProductNotLicensed	This product is not licensed to perform this operation.
NotImplemented	This function has not been implemented on this platform.
UnknownLocalhost	Unable to determine local host name.
InvalidHostAddress	Invalid host address specified.
InvalidServicePort	Invalid service port number specified.
InvalidServiceName	Invalid or unknown service name specified.
InvalidEventId	Invalid event identifier specified.
OperationNotBlocking	No blocking operation in progress on this socket.
SecurityNotInitialized	Unable to initialize security interface for this process.
SecurityContext	Unable to establish security context for this session.
SecurityCredentials	Unable to open certificate store or establish security credentials.
SecurityCertificate	Unable to validate the certificate chain for this session.
SecurityDecryption	Unable to decrypt data stream.
SecurityEncryption	Unable to encrypt data stream.

InvalidUrlScheme	The scheme for the specified URL is invalid or not supported
InvalidUrlHost	The host name for the specified URL is invalid
InvalidUrlPort	The port number for the specified URL is invalid
InvalidUrlPath	The resource path for the specified URL is invalid
InvalidCharset	The specified character set is invalid or not supported
InvalidCodePage	The specified code page is invalid or not supported

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#) | [LastError Property](#) | [LastErrorString Property](#) | [OnError Event](#)

SocketWrench.SecureCipherAlgorithm Enumeration

The supported encryption algorithms.

This enumeration has a `FlagsAttribute` attribute that allows a bitwise combination of its member values.

[Visual Basic]

<Flags>

Public Enum SocketWrench.SecureCipherAlgorithm

[C#]

[Flags]

public enum SocketWrench.SecureCipherAlgorithm

Remarks

The **SecureCipherAlgorithm** enumeration identifies which encryption algorithm was selected when a secure connection was established with the server.

Members

Member Name	Description	Value
None	No cipher has been selected. A secure connection has not been established with the server.	0
Rc2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40- and 128-bits in length, in 8-bit increments.	1
Rc4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40- and 128-bits in length, in 8-bit increments.	2
Rc5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.	4
Des	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher using 56-bit keys.	8
Des3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively using a 168-bit key length.	16
DesX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.	32
Aes	The Advanced Encryption Standard	64

	cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 SP3 and later versions of the operating system.	
Skipjack	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.	128
Blowfish	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.	256

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#)

SocketWrench.SecureHashAlgorithm Enumeration

The supported hash algorithms.

This enumeration has a `FlagsAttribute` attribute that allows a bitwise combination of its member values.

[Visual Basic]

<Flags>

Public Enum SocketWrench.SecureHashAlgorithm

[C#]

[Flags]

public enum SocketWrench.SecureHashAlgorithm

Remarks

The **SecureHashAlgorithm** enumeration identifies the message digest algorithm that was selected when a secure connection was established with the server.

Members

Member Name	Description	Value
None	No hash algorithm has been selected. This is not a secure connection with the server.	0
Md5	The MD5 algorithm was selected. This algorithm produces a 128-bit message digest. This algorithm is no longer considered to be cryptographically secure.	1
Sha1	The SHA-1 algorithm was selected. This algorithm produces a 160-bit message digest. This algorithm is no longer considered to be cryptographically secure.	2
Sha256	The SHA-256 algorithm was selected. This algorithm produces a 256-bit message digest.	4
Sha384	The SHA-384 algorithm was selected. This algorithm produces a 384-bit message digest.	8
Sha512	The SHA-512 algorithm was selected. This algorithm produces a 512-bit message digest.	16

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

SocketWrench.SecureKeyAlgorithm Enumeration

The supported key exchange algorithms.

This enumeration has a `FlagsAttribute` attribute that allows a bitwise combination of its member values.

[Visual Basic]

<Flags>

Public Enum SocketWrench.SecureKeyAlgorithm

[C#]

[Flags]

public enum SocketWrench.SecureKeyAlgorithm

Remarks

The **SecureKeyAlgorithm** enumeration identifies the key exchange algorithm that was selected when a secure connection was established with the server.

Members

Member Name	Description	Value
None	No key exchange algorithm has been selected. This is not a secure connection with the server.	0
Rsa	The RSA key exchange algorithm has been selected.	1
KeaDh	The KEA-DH key exchange algorithm has been selected. This is an improved version of the Diffie-Hellman algorithm.	2
DiffieHellman	The Diffie-Hellman key exchange algorithm has been selected.	4
EllipticCurveDh	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.	8

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#)

SocketWrench.SecurityCertificate Enumeration

The security certificate status.

[Visual Basic]

```
Public Enum SocketWrench.SecurityCertificate
```

[C#]

```
public enum SocketWrench.SecurityCertificate
```

Remarks

The **SecurityCertificate** enumeration identifies the current status of the certificate provided by the server when a secure connection was established.

Members

Member Name	Description
None	No certificate information is available. A secure connection was not established with the server.
Valid	The certificate is valid.
NoMatch	The certificate is valid, however the domain name specified in the certificate does not match the domain name of the server. The application can examine the CertificateSubject property to determine the site the certificate was issued to.
Expired	The certificate has expired and is no longer valid. The application can examine the CertificateExpires property to determine when the certificate expired.
Revoked	The certificate has been revoked and is no longer valid. It is recommended that the application immediately terminate the connection if this status is returned.
Untrusted	The certificate has not been issued by a trusted authority, or the certificate is not trusted on the local host. It is recommended that the application immediately terminate the connection if this status is returned.
Invalid	The certificate is invalid. This typically indicates that the internal structure of the certificate is damaged. It is recommended that the application immediately terminate the connection if this status is returned.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#)

SocketWrench.ShutdownOptions Enumeration

The supported socket shutdown options.

This enumeration has a `FlagsAttribute` attribute that allows a bitwise combination of its member values.

[Visual Basic]

<Flags>

Public Enum SocketWrench.ShutdownOptions

[C#]

[Flags]

public enum SocketWrench.ShutdownOptions

Remarks

The `SocketWrench` class uses the **ShutdownOptions** enumeration to specify how reading and writing on the socket should be handled when the **Shutdown** method is called.

Members

Member Name	Description	Value
Read	Disable any further reading of data. The application will be able to continue to send data. The remote host will see this as the connection being closed.	0
Write	Disable any further sending of data. The application will be able to continue to read data until the remote host closes the connection.	1
ReadWrite	Disable any further reading or writing to the socket. The remote host will see this as the connection being closed.	2

Requirements

Namespace: [SocketTools](#)

Assembly: `SocketTools.SocketWrench` (in `SocketTools.SocketWrench.dll`)

See Also

[SocketTools Namespace](#)

SocketWrench.SocketByteOrder Enumeration

The byte-order in which integer data may exchanged with a remote host.

[Visual Basic]

Public Enum SocketWrench.SocketByteOrder

[C#]

public enum SocketWrench.SocketByteOrder

Remarks

The byte-order is used to specify how 16-bit and 32-bit integer data is written to and read from the socket.

Members

Member Name	Description
Native	Integer data will be sent and received using the native byte order.
Network	Integer data will be sent and received using network byte order.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#)

SocketWrench.SocketOptions Enumeration

The socket options supported by the class.

This enumeration has a FlagsAttribute attribute that allows a bitwise combination of its member values.

[Visual Basic]

<Flags>

Public Enum SocketWrench.SocketOptions

[C#]

[Flags]

public enum SocketWrench.SocketOptions

Members

Member Name	Description	Value
None	No option specified.	0
Broadcast	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.	1
DontRoute	This option specifies default routing should not be used. This option should not be specified unless absolutely necessary.	2
KeepAlive	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This option is only valid for stream sockets.	4
ReuseAddress	This option specifies the local address can be reused. This option is commonly used by server applications.	8
NoDelay	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.	16
InLine	This option specifies that out-of-band data should be received inline with the standard data stream. This option is only valid for stream sockets.	32
TrustedSite	This option specifies the sever should be trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects secure client connections.	2048
Secure	This option specifies that a secure, encrypted connection will be established with the remote host.	4096

SecureFallback	This option specifies the class should permit the use of less secure cipher suites for compatibility with legacy clients and servers. If this option is specified, it will enable connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.	32768
PreferIPv6	This option specifies the client should prefer the use of IPv6 if the remote hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.	262144
FreeThread	This option specifies that class methods may be called from any thread, and not only the thread that established the connection. Using this option disables certain internal safety checks that are made by the class and may result in unexpected behavior unless you ensure that access to the class instance is synchronized across multiple threads.	524288

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#)

SocketWrench.SocketProtocol Enumeration

The supported socket protocol types.

```
[Visual Basic]
Public Enum SocketWrench.SocketProtocol
```

```
[C#]
public enum SocketWrench.SocketProtocol
```

Remarks

The SocketWrench class uses the **SocketProtocol** enumeration to specify which network protocol will be used when a socket is created. The default protocol used by the class is **Stream**.

Members

Member Name	Description
Stream	Transmission Control Protocol (TCP). This protocol should be used with stream sockets, where data is sent and received as an arbitrary stream of bytes.
Datagram	User Datagram Protocol (UDP). This protocol should be used with datagram sockets, where data is sent and received in discrete packets.
Raw	Raw sockets. This socket type is for special purpose applications which need access to the IP datagram.

Requirements

- Namespace:** [SocketTools](#)
- Assembly:** SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#)

SocketWrench.SocketStatus Enumeration

The current state of the socket.

```
[Visual Basic]
Public Enum SocketWrench.SocketStatus
```

```
[C#]
public enum SocketWrench.SocketStatus
```

Members

Member Name	Description
Unused	A socket has not been created. Attempts to perform any network operations, such as sending or receiving data, will generate an error.
Idle	A socket has been created, but is not currently in use. A blocking socket operation can be executed without interfering with any other requests.
Listen	The socket is listening for connections.
Connect	The socket is in the process of establishing a connection.
Accept	The socket is in the process of accepting a connection.
Read	The socket is in the process of receiving data.
Write	The socket is in the process of sending data.
Flush	The socket buffers are in the process of being flushed. Any data in the receive buffers will be discarded.
Disconnect	The socket is being closed and subsequent attempts to access the socket will result in an error.

Requirements

- Namespace:** [SocketTools](#)
- Assembly:** SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#)

SocketWrench.SocketStream Enumeration

The supported data streaming options.

```
[Visual Basic]
Public Enum SocketWrench.SocketStream
```

```
[C#]
public enum SocketWrench.SocketStream
```

Remarks

The SocketWrench class uses the **SocketStream** enumeration to specify how data should be processed when read from a socket using either the **ReadStream** or **StoreStream** methods.

Members

Member Name	Description
Default	The data stream will be returned to the caller unmodified. This option should always be used with binary data or data being stored in a byte array. If no options are specified, this is the default option used by this method.
Convert	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.

Requirements

- Namespace:** [SocketTools](#)
- Assembly:** SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#) | [ReadStream Method](#) | [StoreStream Method](#)

SocketWrench.TraceOptions Enumeration

The supported trace logging options.

This enumeration has a `FlagsAttribute` attribute that allows a bitwise combination of its member values.

[Visual Basic]

<Flags>

Public Enum SocketWrench.TraceOptions

[C#]

[Flags]

public enum SocketWrench.TraceOptions

Remarks

The **TraceOptions** enumeration determines what kind of information is written to the logfile specified by the **TraceFile** property. These options are only meaningful when logging is enabled by setting the **Trace** property to **true**.

Members

Member Name	Description	Value
Info	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.	0
Error	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.	1
Warning	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.	2
HexDump	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload	4

	contents and responses.	
Truncate	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.	8
Process	All network function calls made by the current process are logged, rather than only those functions in the current thread. This option is useful for multithreaded applications that are using worker threads.	4096
Default	The same as specifying the HexDump option.	4

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#)

SocketWrench.ThreadModelAttribute Class

Attribute that defines the threading model for the class.

For a list of all members of this type, see [SocketWrench.ThreadModelAttribute Members](#).

System.Object

System.Attribute

SocketTools.SocketWrench.ThreadModelAttribute

[Visual Basic]

```
<AttributeUsage(ValidOn:=AttributeTargets.Assembly, AllowMultiple:=False, Inherited:=True)>  
Public Class SocketWrench.ThreadModelAttribute  
    Inherits Attribute
```

[C#]

```
[AttributeUsage(ValidOn=AttributeTargets.Assembly, AllowMultiple=False, Inherited=True)]  
public class SocketWrench.ThreadModelAttribute : Attribute
```

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

The ThreadModel attribute is used to define the threading model that is to be used when an instance of the class is created. This attribute is defined in the assembly information module for the language, such as AssemblyInfo.cs when programming C#.

Example

```
<Assembly:  
SocketTools.SocketWrench.ThreadModel(SocketTools.SocketWrench.ThreadModelAttribute.Model.SingleThread)>
```

```
[assembly:  
SocketTools.SocketWrench.ThreadModel(SocketTools.SocketWrench.ThreadModelAttribute.Model.SingleThread)]
```

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketWrench.ThreadModelAttribute Members](#) | [SocketTools Namespace](#) | [ThreadModel Property](#)
([SocketTools.SocketWrench](#))

SocketWrench.ThreadModelAttribute Members

[SocketWrench.ThreadModelAttribute overview](#)

Public Instance Constructors

 SocketWrench.ThreadModelAttribute Constructor	Constructor for the ThreadModel attribute which defines the threading model.

Public Instance Properties

ThreadModel	Returns the threading model used by the class.
TypeId (inherited from Attribute)	When implemented in a derived class, gets a unique identifier for this Attribute.

Public Instance Methods

Equals (inherited from Attribute)	
GetHashCode (inherited from Attribute)	Returns the hash code for this instance.
GetType (inherited from Object)	Gets the Type of the current instance.
IsDefaultAttribute (inherited from Attribute)	When overridden in a derived class, returns an indication whether the value of this instance is the default value for the derived class.
Match (inherited from Attribute)	When overridden in a derived class, returns a value indicating whether this instance equals a specified object.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[SocketWrench.ThreadModelAttribute Class](#) | [SocketTools Namespace](#) | [ThreadModel Property](#)
([SocketTools.SocketWrench](#))

SocketWrench.ThreadModelAttribute Constructor

Constructor for the ThreadModel attribute which defines the threading model.

[Visual Basic]

```
Public Sub New( _  
    ByVal threadModel As Model _  
)
```

[C#]

```
public SocketWrench.ThreadModelAttribute(  
    Model threadModel  
);
```

Parameters

threadModel

A [Model](#) enumeration value which specifies the threading model which will be used when creating an instance of the class. A value of zero specifies a single threaded model, while a non-zero value specifies a free threaded model.

Remarks

The **ThreadModel** attribute specifies the threading model that is used by the class instance when a connection is established. The default threading model is single threaded, which specifies that only the thread that established the connection should be permitted to invoke methods.

It is important to note that the single threading model does not limit the application to a single thread of execution. When a session is established using the **Connect** method, that session is attached to the thread that created it. From that point on, until the session is terminated, only the owner may invoke methods in that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Setting this attribute to a non-zero value disables certain internal safety checks that are performed by the class and may result in unexpected behavior unless you ensure that access to the class instance is synchronized across multiple threads. The application must ensure that no two threads will attempt to invoke a blocking method at the same time. In other words, if one thread invokes a method, the application must ensure that another thread will not attempt to invoke any other method at the same time using the same instance of the class.

Changing the value of this attribute will also change the default value for the **ThreadModel** property for all instances of the class.

See Also

[SocketWrench.ThreadModelAttribute Class](#) | [SocketTools Namespace](#)

SocketWrench.ThreadModelAttribute Properties

The properties of the **SocketWrench.ThreadModelAttribute** class are listed below. For a complete list of **SocketWrench.ThreadModelAttribute** class members, see the [SocketWrench.ThreadModelAttribute Members](#) topic.

Public Instance Properties

ThreadModel	Returns the threading model used by the class.
TypeId (inherited from Attribute)	When implemented in a derived class, gets a unique identifier for this Attribute.

See Also

[SocketWrench.ThreadModelAttribute Class](#) | [SocketTools Namespace](#) | [ThreadModel Property \(SocketTools.SocketWrench\)](#)

SocketWrench.ThreadModelAttribute.ThreadModel Property

Returns the threading model used by the class.

[Visual Basic]

```
Public Property ThreadModel As Model
```

[C#]

```
public SocketWrench.ThreadModelAttribute.Model ThreadModel {get; set;}
```

Property Value

A [Model](#) enumeration value which specifies the threading model which will be used when an instance of the class is created.

See Also

[SocketWrench.ThreadModelAttribute Class](#) | [SocketTools Namespace](#)

SocketWrench.ThreadModelAttribute.Model Enumeration

Specifies the threading model used by the class instance.

```
[Visual Basic]
Public Enum SocketWrench.ThreadModelAttribute.Model

[C#]
public enum SocketWrench.ThreadModelAttribute.Model
```

Remarks

The threading model **SingleThread** does not limit the application to a single thread of execution. It specifies that only a single thread may invoke methods in a class instance. When a session is established using the **Connect** method, that session is attached to the thread that created it. From that point on, until the session is terminated, only the owner may invoke methods in that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

The threading model **FreeThread** disables certain internal safety checks that are performed by the class and may result in unexpected behavior unless you ensure that access to the class instance is synchronized across multiple threads. The application must ensure that no two threads will attempt to invoke a blocking method at the same time. In other words, if one thread invokes a method, the application must ensure that another thread will not attempt to invoke any other method at the same time using the same instance of the class.

Members

Member Name	Description
SingleThread	Methods in the class instance may only be invoked by a single thread. This threading model specifies that only the thread which established the connection should be permitted to invoke methods. This is the default threading model.
FreeThread	Methods in the class instance may be invoked by any thread. This threading model permits methods to be invoked across multiple threads without being explicitly attached to the object.

Requirements

- Namespace:** [SocketTools](#)
- Assembly:** SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketTools Namespace](#)

SocketWrench.RuntimeLicenseAttribute Class

Attribute that defines the runtime license key for the class.

For a list of all members of this type, see [SocketWrench.RuntimeLicenseAttribute Members](#).

System.Object

System.Attribute

SocketTools.SocketWrench.RuntimeLicenseAttribute

[Visual Basic]

```
<AttributeUsage(ValidOn:=AttributeTargets.Assembly, AllowMultiple:=False,
    Inherited:=True)>
```

```
Public Class SocketWrench.RuntimeLicenseAttribute
    Inherits Attribute
```

[C#]

```
[AttributeUsage(ValidOn=AttributeTargets.Assembly, AllowMultiple=False,
    Inherited=True)]
```

```
public class SocketWrench.RuntimeLicenseAttribute : Attribute
```

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

The RuntimeLicense attribute is used to define the runtime license key that will be used when an instance of the class is created. This attribute is defined in the assembly information module for the language, such as AssemblyInfo.cs when programming C#. The runtime license key must be defined if you wish to redistribute your application.

The runtime license key for your copy of SocketWrench can be generated using the License Manager utility that is included with the product. Note that if you have installed an evaluation license, you will not have a runtime license key and cannot redistribute any applications which use the SocketWrench class.

Example

```
<Assembly: SocketTools.SocketWrench.RuntimeLicense("abcdefghijklmnop")>
```

```
[assembly: SocketTools.SocketWrench.RuntimeLicense("abcdefghijklmnop")]
```

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketWrench.RuntimeLicenseAttribute Members](#) | [SocketTools Namespace](#)

SocketWrench.RuntimeLicenseAttribute Members

[SocketWrench.RuntimeLicenseAttribute overview](#)

Public Instance Constructors

 SocketWrench.RuntimeLicenseAttribute Constructor	Constructor for the RuntimeLicense attribute which defines the runtime license key.

Public Instance Properties

LicenseKey	Returns the value of the runtime license key.
TypeId (inherited from Attribute)	When implemented in a derived class, gets a unique identifier for this Attribute.

Public Instance Methods

Equals (inherited from Attribute)	
GetHashCode (inherited from Attribute)	Returns the hash code for this instance.
GetType (inherited from Object)	Gets the Type of the current instance.
IsDefaultAttribute (inherited from Attribute)	When overridden in a derived class, returns an indication whether the value of this instance is the default value for the derived class.
Match (inherited from Attribute)	When overridden in a derived class, returns a value indicating whether this instance equals a specified object.
ToString (inherited from Object)	Returns a String that represents the current Object.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[SocketWrench.RuntimeLicenseAttribute Class](#) | [SocketTools Namespace](#)

SocketWrench.RuntimeLicenseAttribute Constructor

Constructor for the RuntimeLicense attribute which defines the runtime license key.

[Visual Basic]

```
Public Sub New( _  
    ByVal licenseKey As String _  
)
```

[C#]

```
public SocketWrench.RuntimeLicenseAttribute(  
    string licenseKey  
);
```

Parameters

licenseKey

A string argument which specifies the runtime license key which will be used to initialize the class library.

Remarks

The runtime license key for your copy of SocketWrench can be generated using the License Manager utility that is included with the product. Note that if you have installed an evaluation license, you will not have a runtime license key and cannot redistribute any applications which use the SocketWrench class.

See Also

[SocketWrench.RuntimeLicenseAttribute Class](#) | [SocketTools Namespace](#)

SocketWrench.RuntimeLicenseAttribute Properties

The properties of the **SocketWrench.RuntimeLicenseAttribute** class are listed below. For a complete list of **SocketWrench.RuntimeLicenseAttribute** class members, see the [SocketWrench.RuntimeLicenseAttribute Members](#) topic.

Public Instance Properties

LicenseKey	Returns the value of the runtime license key.
TypeId (inherited from Attribute)	When implemented in a derived class, gets a unique identifier for this Attribute.

See Also

[SocketWrench.RuntimeLicenseAttribute Class](#) | [SocketTools Namespace](#)

SocketWrench.RuntimeLicenseAttribute.LicenseKey Property

Returns the value of the runtime license key.

[Visual Basic]

Public Property LicenseKey As String

[C#]

public string LicenseKey {get; set;}

Property Value

A string which contains the runtime license key.

See Also

[SocketWrench.RuntimeLicenseAttribute Class](#) | [SocketTools Namespace](#)

SocketWrenchException Class

The exception that is thrown when a socket error occurs.

For a list of all members of this type, see [SocketWrenchException Members](#).

System.Object

System.Exception

System.ApplicationException

SocketTools.SocketWrenchException

[Visual Basic]

Public Class SocketWrenchException

Inherits ApplicationException

[C#]

public class SocketWrenchException : ApplicationException

Thread Safety

Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.

Remarks

A SocketWrenchException is thrown by the SocketWrench class when an error occurs.

The default constructor for the SocketWrenchException class sets the **ErrorCode** property to the last socket error that occurred.

Requirements

Namespace: [SocketTools](#)

Assembly: SocketTools.SocketWrench (in SocketTools.SocketWrench.dll)

See Also

[SocketWrenchException Members](#) | [SocketTools Namespace](#)

SocketWrenchException Members

[SocketWrenchException overview](#)

Public Instance Constructors

 SocketWrenchException	Overloaded. Initializes a new instance of the SocketWrenchException class.

Public Instance Properties

ErrorCode	Gets a value which specifies the error that caused the exception.
HelpLink (inherited from Exception)	Gets or sets a link to the help file associated with this exception.
InnerException (inherited from Exception)	Gets the Exception instance that caused the current exception.
Message	Gets a value which describes the error that caused the exception.
Number	Gets a value which specifies the numeric value of the error that caused the exception.
Source (inherited from Exception)	Gets or sets the name of the application or the object that causes the error.
StackTrace (inherited from Exception)	Gets a string representation of the frames on the call stack at the time the current exception was thrown.
TargetSite (inherited from Exception)	Gets the method that throws the current exception.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetBaseException (inherited from Exception)	When overridden in a derived class, returns the Exception that is the root cause of one or more subsequent exceptions.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetObjectData (inherited from Exception)	When overridden in a derived class, sets the SerializationInfo with information about the exception.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString	Creates and returns a string representation of the current exception.

Protected Instance Properties

 HRESULT (inherited from Exception)	Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[SocketWrenchException Class](#) | [SocketTools Namespace](#)

SocketWrenchException Constructor

Initializes a new instance of the [SocketWrenchException](#) class with the last network error code.

Overload List

Initializes a new instance of the [SocketWrenchException](#) class with the last network error code.

```
public SocketWrenchException();
```

Initializes a new instance of the [SocketWrenchException](#) class with a specified error number.

```
public SocketWrenchException(int);
```

Initializes a new instance of the [SocketWrenchException](#) class with a specified error message.

```
public SocketWrenchException(string);
```

Initializes a new instance of the [SocketWrenchException](#) class with a specified error message and a reference to the inner exception that is the cause of this exception.

```
public SocketWrenchException(string,Exception);
```

See Also

[SocketWrenchException Class](#) | [SocketTools Namespace](#)

SocketWrenchException Constructor ()

Initializes a new instance of the [SocketWrenchException](#) class with the last network error code.

[Visual Basic]

Overloads Public Sub New()

[C#]

public SocketWrenchException();

Remarks

The ctor constructor sets the **ErrorCode** property to the last socket error that occurred. For more information about the errors that may occur, refer to the [SocketWrench.ErrorCode](#) enumeration.

See Also

[SocketWrenchException Class](#) | [SocketTools Namespace](#) | [SocketWrenchException Constructor Overload List](#)

SocketWrenchException Constructor (String)

Initializes a new instance of the [SocketWrenchException](#) class with a specified error message.

[Visual Basic]

```
Overloads Public Sub New( _  
    ByVal message As String _  
)
```

[C#]

```
public SocketWrenchException(  
    string message  
);
```

Parameters

message

The error message that explains the reason for the exception.

Remarks

The content of the *message* parameter is intended to be understood by humans. The caller of this constructor is required to ensure that this string has been localized for the current system culture.

See Also

[SocketWrenchException Class](#) | [SocketTools Namespace](#) | [SocketWrenchException Constructor Overload List](#)

SocketWrenchException Constructor (String, Exception)

Initializes a new instance of the [SocketWrenchException](#) class with a specified error message and a reference to the inner exception that is the cause of this exception.

[Visual Basic]

```
Overloads Public Sub New( _  
    ByVal message As String, _  
    ByVal innerException As Exception _  
)
```

[C#]

```
public SocketWrenchException(  
    string message,  
    Exception innerException  
);
```

Parameters

message

The error message that explains the reason for the exception.

innerException

The exception that is the cause of the current exception. If the *innerException* parameter is not a null reference, the current exception is raised in a **catch** block that handles the inner exception.

Remarks

The content of the message parameter is intended to be understood by humans. The caller of this constructor is required to ensure that this string has been localized for the current system culture.

An exception that is thrown as a direct result of a previous exception should include a reference to the previous exception in the **InnerException** property. The **InnerException** property returns the same value that is passed into the constructor, or a null reference if the **InnerException** property does not supply the inner exception value to the constructor.

See Also

[SocketWrenchException Class](#) | [SocketTools Namespace](#) | [SocketWrenchException Constructor Overload List](#)

SocketWrenchException Constructor (Int32)

Initializes a new instance of the [SocketWrenchException](#) class with a specified error number.

[Visual Basic]

```
Overloads Public Sub New( _  
    ByVal code As Integer _  
)
```

[C#]

```
public SocketWrenchException(  
    int code  
);
```

Parameters

code

An integer value which specifies an error code.

Remarks

This constructor sets the **ErrorCode** property to the specified error code. For more information about the errors that may occur, refer to the [SocketWrench.ErrorCode](#) enumeration.

See Also

[SocketWrenchException Class](#) | [SocketTools Namespace](#) | [SocketWrenchException Constructor Overload List](#)

SocketWrenchException Properties

The properties of the **SocketWrenchException** class are listed below. For a complete list of **SocketWrenchException** class members, see the [SocketWrenchException Members](#) topic.

Public Instance Properties

ErrorCode	Gets a value which specifies the error that caused the exception.
HelpLink (inherited from Exception)	Gets or sets a link to the help file associated with this exception.
InnerException (inherited from Exception)	Gets the Exception instance that caused the current exception.
Message	Gets a value which describes the error that caused the exception.
Number	Gets a value which specifies the numeric value of the error that caused the exception.
Source (inherited from Exception)	Gets or sets the name of the application or the object that causes the error.
StackTrace (inherited from Exception)	Gets a string representation of the frames on the call stack at the time the current exception was thrown.
TargetSite (inherited from Exception)	Gets the method that throws the current exception.

Protected Instance Properties

 HRESULT (inherited from Exception)	Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.

See Also

[SocketWrenchException Class](#) | [SocketTools Namespace](#)

SocketWrenchException.ErrorCode Property

Gets a value which specifies the error that caused the exception.

[Visual Basic]

```
Public ReadOnly Property ErrorCode As ErrorCode
```

[C#]

```
public SocketWrench.ErrorCode ErrorCode {get;}
```

Property Value

Returns a [SocketWrench.ErrorCode](#) enumeration value which specifies the error code.

Remarks

The **ErrorCode** property returns the error code that specifies the cause of the exception.

The default constructor for the [SocketWrenchException](#) class sets the error code to the last network error that occurred. For more information about the errors that may occur, refer to the [SocketWrench.ErrorCode](#) enumeration.

See Also

[SocketWrenchException Class](#) | [SocketTools Namespace](#)

SocketWrenchException.Message Property

Gets a value which describes the error that caused the exception.

[Visual Basic]

Overrides Public ReadOnly Property Message As String

[C#]

public override string Message {get;}

Property Value

A string which describes the error that caused the exception.

Remarks

The **Message** property returns a string which describes the error that caused the exception.

See Also

[SocketWrenchException Class](#) | [SocketTools Namespace](#)

SocketWrenchException.Number Property

Gets a value which specifies the numeric value of the error that caused the exception.

[Visual Basic]

Public ReadOnly Property Number As Integer

[C#]

public int Number {get;}

Property Value

An integer value that specifies the error that caused the exception.

Remarks

The **Number** property returns an integer value which specifies the numeric value of the error that caused the exception. This value is the same as the values returned by the Windows Sockets API. For more information about socket error codes, see the Windows Socket Version 2 API error code documentation in MSDN.

See Also

[SocketWrenchException Class](#) | [SocketTools Namespace](#)

SocketWrenchException Methods

The methods of the **SocketWrenchException** class are listed below. For a complete list of **SocketWrenchException** class members, see the [SocketWrenchException Members](#) topic.

Public Instance Methods

Equals (inherited from Object)	Determines whether the specified Object is equal to the current Object.
GetBaseException (inherited from Exception)	When overridden in a derived class, returns the Exception that is the root cause of one or more subsequent exceptions.
GetHashCode (inherited from Object)	Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetObjectData (inherited from Exception)	When overridden in a derived class, sets the SerializationInfo with information about the exception.
GetType (inherited from Object)	Gets the Type of the current instance.
ToString	Creates and returns a string representation of the current exception.

Protected Instance Methods

 Finalize (inherited from Object)	Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
 MemberwiseClone (inherited from Object)	Creates a shallow copy of the current Object.

See Also

[SocketWrenchException Class](#) | [SocketTools Namespace](#)

SocketWrenchException.ToString Method

Creates and returns a string representation of the current exception.

[Visual Basic]

Overrides Public Function ToString() As String

[C#]

public override string ToString();

Return Value

A string representation of the current exception.

Remarks

The **ToString** method returns a representation of the current exception that is intended to be understood by humans. Where the exception contains culture-sensitive data, the string representation returned by **ToString** is required to take into account the current system culture. Although there are no exact requirements for the format of the returned string, it should attempt to reflect the value of the object as perceived by the user.

This implementation of **ToString** obtains the numeric error code value and a description of the error that caused the current exception. If there is no error message or it is an empty string, then no error message is returned.

This method overrides **ApplicationException.ToString**.

See Also

[SocketWrenchException Class](#) | [SocketTools Namespace](#)

SocketWrench 12 Control Overview

The SocketWrench software development kit includes an ActiveX control which can be used in a variety of legacy programming languages such as Visual Basic 6.0, Visual FoxPro and Visual Basic for Applications (VBA). The control implements the same general set of features and functions that are available in the native library API. However, the control provides a simpler programming interface.

To include the SocketWrench control in your project in Visual Basic, simply select the **Project|Components** menu option and select the SocketWrench Control. In other languages, follow the normal steps that are taken to include an ActiveX control in your development project.

Commonly Used Properties

Although SocketWrench has a large number of properties, only a subset of them will be used with any frequency in your applications. Here are the properties that you should become familiar with first:

Property	Description
Blocking	Gets and sets the blocking mode for the socket. By default, sockets are blocking, and no socket events will be generated by the control.
HostAddress	Specifies the IP address of the remote host that the socket will be sending data to or receiving data from. Setting this property will automatically update the HostName property.
HostName	Specifies the name of the remote host. Setting this property will automatically update the HostAddress property.
Protocol	Specifies the protocol to use for this socket. Typical values are swProtocolTcp and swProtocolUdp .
RemotePort	Specifies the port number that a server is listening. Setting this value causes the RemoteService property to be updated.

Commonly Used Methods

SocketWrench has a number of methods which are used to perform some function, such as establishing a connection or sending and receiving data. In many cases, there are optional arguments that can be passed to these methods. If an argument is omitted, the value of a previously set property may be used. If no value has been specified, then a reasonable default value is typically used.

Methods	Description
Connect	This method is used to establish a connection with a server. This method returns 0 if the connection attempt was successful, or an error code which indicates the cause of the failure.
Disconnect	This method is used to terminate a connection with a server. This method returns 0 if the connection attempt was successful, or an error code which indicates the cause of the failure.
Listen	This method causes the control to listen on a socket for incoming connections on the port specified by the LocalPort property. If a socket has not already been created, this method will create it. This method returns 0 if the connection attempt was successful, or an error code which indicates the cause of the failure.
Read	This method reads the specified number of bytes into a string buffer. The number of bytes actually read is returned. A return value of 0 indicates that the remote host has closed the

	socket connection, and a return value of -1 indicates that an error has occurred.
Write	This method writes the specified number of bytes from a string buffer to the socket. The number of bytes actually written is returned. A return value of -1 indicates that an error has occurred.

Control Initialization

When you begin developing your application using the SocketWrench control, the first thing that must happen is the control must be initialized. In some development environments, such as Visual Basic, this is done automatically when the control is inserted into a form. In other languages, this must be done explicitly by calling the Initialize method for each instance of the control.

The initialization method serves two purposes. It loads the Windows networking libraries required to establish a connection and it validates the runtime license key that you provide. The runtime license key is a string of characters which identifies your license to use and redistribute the SocketWrench controls. It is unique to your product serial number and must be used when redistributing your application to an end-user. Many languages will handle the licensing issue transparently, however some languages may require that you explicitly provide your runtime licensing key.

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

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

The runtime license key is normally stored in the Include folder where you installed SocketWrench and is defined in a file named "cswskey8" which can be included with your application. For example, C/C++ programmers would use the cswskey11.h header file while Visual Basic programmers would use the cswskey11.bas module. The Visual Basic module would define the runtime license key as:

```
,
' SocketWrench 12.0
' Copyright 2026 Catalyst Development Corporation
' All rights reserved
,
' This file is licensed to you pursuant to the terms of the
' product license agreement included with the original software
' and is protected by copyright law and international treaties.
,

Public Const CSWSOCK12_LICENSE_KEY As String = ""
```

This could either be included with your Visual Basic application or you could simply copy the string into your application. The control could then be initialized like this:

```
,
' Initialize the control using the specified runtime
' license key; if the key is not specified, the
' development license will be used
,

nError = ctlSocket.Initialize(CSWSOCK12_LICENSE_KEY)
If nError > 0 Then
    MsgBox "Unable to initialize SocketWrench component"
End
End If
```

If the Initialize method fails, it will return an error code value that indicates the reason for the failure. A return value of zero indicates that the control was initialized successfully.

An application is only required to call a control's initialization method once, but it must be called for each instance of the control that is used. It is safe to call the initialization method more than once, but for each time that it is called, you must call the Uninitialize method for that control before your program terminates. In other words, if you called Initialize at the beginning of your program, you must call Uninitialize before your program ends. The Uninitialize method performs any necessary housekeeping operations, such as returning memory allocated for the control back to the operating system. If there are any open connections at the time that the Uninitialize method is called, they will be aborted. After the control has been uninitialized, you must call the Initialize method again in order to use any of the control's other methods.

Internet Dialer Control

Create and monitor dial-up networking connections to an Internet service provider.

Reference

- [Properties](#)
- [Methods](#)
- [Events](#)
- [Error Codes](#)

Control Information

Object Name	RasDialerCtl.Dialer
File Name	CSRASX12.OCX
Version	12.0.3024.1286
ProgID	SocketTools.Dialer.12
ClassID	5D71D87F-A6B8-4E18-B392-4DA030E629F3
Threading Model	Apartment
Help File	CSW12HLP.CHM
Dependencies	None
Standards	RFC 1055, RFC 1661

Overview

This control provides a way for client applications to connect to a server using Microsoft Windows Remote Access Services (RAS). To use this control, the dial-up networking software must be installed on the local system. For access to the Internet, the TCP/IP protocol must be installed and configured. The control may be configured to use either the SLIP or PPP protocols, depending on the requirements of the service provider. Refer to your system documentation for information about installing and configuring dial-up networking on your system.

For those applications which may be used in a mobile environment, or otherwise require remote network access, the Dialer control provides a convenient interface to this service. Connections can be established and discontinued under the direct control of the program, rather than requiring that the user execute another program before starting your application.

Requirements

The SocketTools ActiveX Edition components are self-registering controls that are compatible with any programming language that supports the Component Object Model (COM) and the ActiveX control specification. If you are using Visual Basic 6.0, Service Pack 6 (SP6) is required. We recommend installing all available updates for your development tools. Although the Visual Basic 6.0 runtime is supported on Windows 11, the IDE itself is no longer officially supported by Microsoft. Legacy IDEs and compilers may also require elevated (administrator) privileges to run properly.

The ActiveX controls are supported on Windows 7, Windows Server 2008 R2, and later versions of both desktop and server editions of Windows. For Windows 7, Service Pack 1 (SP1) is the minimum requirement. For Windows 8 systems, the 8.1 update must be installed. On ARM-based platforms, the

minimum supported version is Windows 10 build 1709.

SocketTools includes components for both 32-bit (x86) and 64-bit (x64) Windows platforms. All modern versions of Windows are 64-bit, but we continue to include 32-bit components for compatibility with legacy development tools such as Visual Basic 6.0. We also provide native ARM64 ActiveX controls for systems running Windows on ARM. Note that 32-bit applications on ARM run using x86 emulation, and Microsoft no longer supports native ARM32 applications.

If you are using VBScript, Visual Basic for Applications (VBA), or another scripting environment on an ARM-based system, be aware that 64-bit hosts will require ARM64 components, while 32-bit hosts will use x86 components under emulation. This includes applications created with Visual Basic 6.0. To reduce deployment complexity and avoid confusion, SocketTools does not include ARM32 ActiveX controls in its standard redistribution packages but they are available on request for legacy or specialized scenarios.

Distribution

When you distribute an application that uses this control, you can either install the file in the same folder as your application executable or as a shared component in the appropriate system folder. If you install the control in the system folder, it is important that you distribute the correct version for the target platform and it must be registered. If you install the control in the same folder as your executable, it is recommended that you use registration-free activation or COM redirection to ensure that the correct version of the control is loaded by the application.

Internet Dialer Control Properties

Property	Description
AreaCode	Gets and sets the area code for the current phonebook entry
AutoConnect	Automatically detect connections established by another process
AutoDial	Determine if autodialing has been enabled on the local system
AutoDisconnect	Automatically disconnect from the server when the control is unloaded
Available	Determine if the Remote Access Service is available
Blocking	Gets and sets the blocking state of the control
BytesIn	Returns the number of bytes that have been received by the dial-up networking device
BytesOut	Returns the number of bytes that have been transmitted by the dial-up networking device
Callback	Specifies that the server should call the system back
CallbackNumber	Specifies the telephone number for the server to call back on
Connection	Return the handle for the specified dial-up networking session
Connections	Return the number of active dial-up networking sessions
ConnectSpeed	Returns the line speed for the current dial-up networking connection
CountryCode	Gets and sets the country code for the current phonebook entry
CountryName	Gets and sets the country name for the current phonebook entry
DefaultGateway	Set the default route for IP packets through the dial-up adapter
DeviceCount	Returns the number of dial-up networking devices available
DeviceEntry	Return the name of the specified device entry
DeviceName	Gets and sets the device name for the current dial-up networking connection
DeviceType	Gets and sets the device type for the current dial-up networking connection
DynamicAddress	Configure the current phonebook entry to use a dynamic IP address
DynamicNameServers	Configure the current phonebook entry to use dynamic nameservers
FramingProtocol	Gets and sets the framing protocol for the current phonebook entry
InternetAddress	Return the IP address assigned to the current dial-up networking session
Interval	Gets and sets the interval at which the connection is monitored
IpHeaderCompression	Configure the current phonebook entry to enable IP header compression
IsConnected	Determine if the control is connected to a service provider
IsInitialized	Determine if the control has been initialized
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error that occurred
LcpExtensions	Configure the current phonebook entry to use PPP LCP extensions
LocalNumber	Gets and sets the local phone number specified in the phonebook entry
ModemLights	Enable or disable the dial-up networking system tray icon
ModemSpeaker	Enable or disable the modem speaker

NameServer	Gets and sets the IP addresses of the nameservers assigned to the current phonebook entry
NetworkLogon	Configure the current phonebook entry to logon to the network
NetworkProtocol	Gets and sets the network protocol for the current phonebook entry
Password	The password required to establish a connection with the server
PhoneBook	Sets the file name of the Remote Access phone book to use
PhoneBookEntries	Return the number of entries in the current phone book
PhoneBookEntry	Return the name for the specified phone book entry
PhoneEntry	Specify the phone book entry to use to establish a connection with a server
PhoneNumber	Specifies the telephone number of the server
RequireEncryption	Configure the current phonebook entry to require secure authentication
ScriptFile	Gets and sets the name of the script file for the current phonebook entry
ServerAddress	Return the IP address of the dial-up networking server
SoftwareCompression	Configure the current phonebook entry to negotiate software compression
Status	Return the current status of the control
Terminal	Determine if a terminal window is displayed during the connection process
ThrowError	Enables or disables error handling by the application
Timeout	Gets and sets the number of seconds until a connection attempt fails
UserDomain	Specifies the NT domain on which user authentication is to occur
UserName	Set the user name that is required to establish a connection with the server
UserPhoneBook	Returns the name of the default user phonebook
Version	Return the current version of the control

AreaCode Property

Gets and sets the area code for the current phonebook entry.

Syntax

```
[form].object.AreaCode [= areacode ]
```

Remarks

The **AreaCode** property is used to set or return the current phonebook entry's area code. If no area code has been specified, then this property will return an empty string. The value of this property is ignored unless the **CountryCode** property is also set to a valid country code.

Data Type

String

See Also

[CountryCode Property](#), [CountryName Property](#), [LoadEntry Method](#), [SaveEntry Method](#)

AutoConnect Property

Automatically detect connections established by another process.

Syntax

object.AutoConnect = { True | False }

Remarks

The **AutoConnect** property determines if the control automatically detects if a connection has been established by another process. When enabled, the control will periodically check for any connections that have been established. The **Interval** property controls the frequency in which the control performs this check.

If the control detects that a connection has been made, it will immediately fire the **OnConnect** event, followed by the **OnStatus** event, to indicate that a connection has been established. The control then begins to monitor that connection as usual, until that connection is dropped or the control is unloaded.

To periodically check to see if a connection has been established by another process without using the **AutoConnect** property, read the value of the **Connections** property, which returns the number of active dial-up networking connections. A value greater than zero indicates that a dial-up networking connection has been established.

If there are multiple dial-up networking devices on the system, it may be possible for more than one connection to be active at a time. If this is the case, setting the **AutoConnect** property to True will cause the control to inherit the first active connection. To manage multiple dial-up connections, use the **Connection** property array to enumerate the available connections and set the **Handle** property to take control of a specific session.

Data Type

Boolean

See Also

[AutoDisconnect Property](#), [Connection Property](#), [Connections Property](#), [IsConnected Property](#), [Connect Method](#), [Disconnect Method](#), [Interval Property](#), [OnConnect Event](#), [OnDisconnect Event](#), [OnStatus Event](#)

AutoDial Property

Determine if autodialing has been enabled on the local system.

Syntax

[*form*].*object*.**AutoDial** [= { True | False }]

Remarks

The **AutoDial** property can be used to determine if autodialing is enabled or disabled on the current system. When autodialing is enabled and an application attempts to establish a connection over the Internet, a dialog box will be displayed asking the user if they want to connect to their default service provider. This property will return True if autodialing is currently enabled, or False if it has been disabled.

Setting the **AutoDial** property allows an application to change the autodial settings for the current user. Setting the property value to True specifies that you wish to enable autodialing, and the system will prompt the user to establish a dial-up connection when necessary. Setting the property to False disables autodialing, and prevents the system from prompting the user. This can be beneficial if your application needs to run in an unattended mode. If you change the autodial settings for the user, it is recommended that you restore them to their original value before the application terminates.

This property can only be changed by applications running under Windows 98, Windows NT 4.0 and later versions. If the autodial settings cannot be changed by the current user, an error will be generated.

Data Type

Boolean

AutoDisconnect Property

Automatically disconnect from the server when the control is unloaded.

Syntax

`object.AutoDisconnect = { True | False }`

Remarks

The **AutoDisconnect** property determines if the control should automatically disconnect from a server when the control is unloaded, typically when the application terminates. The default value for this property is True.

If a dial-up connection was already established at the time the control was loaded, this property will be reset to False, preventing it from automatically disconnecting from the host when it is unloaded. Therefore, to always force the control to automatically terminate a connection when it is unloaded, you must explicitly set the property value to True in your application.

Data Type

Boolean

See Also

[AutoConnect Property](#), [IsConnected Property](#), [Connect Method](#), [Disconnect Method](#)

Available Property

Determine if the Remote Access Service is available.

Syntax

object.Available

Remarks

This read-only property returns True if the Remote Access Service (RAS) software has been installed on the system. Note that this property does **not** indicate that the required hardware is available or that a specific protocol has been configured.

Data Type

Boolean

Blocking Property

Gets and sets the blocking state of the control.

Syntax

`[form].object.Blocking` [= { True | False }]

Remarks

The **Blocking** property determines how the control establishes a dial-up connection. If set to True, the control will wait until a connection has been established or the connection attempt fails before returning control to the application. If set to False, the control will begin the connection process and return control immediately to the application. For a non-blocking connection, the application should monitor the **OnStatus** event to determine the progress of the connection attempt. The default value for this property is False.

Data Type

Boolean

See Also

[IsConnected Property](#), [Status Property](#), [Connect Method](#), [OnStatus Event](#)

BytesIn Property

Returns the number of bytes that have been received by the dial-up networking device.

Syntax

`[form].object.BytesIn`

Remarks

The **BytesIn** property returns the number of bytes that have been received by the dial-up networking device. If the control is unable to determine the number of bytes received, it will return a value of zero.

This property is only supported with applications running under Windows 98 and Windows 2000. A general purpose application designed to run on all of the common Windows platforms should expect that this property may return zero as a value.

Data Type

Integer (Int32)

See Also

[BytesOut Property](#), [ConnectSpeed Property](#)

BytesOut Property

Returns the number of bytes that have been transmitted by the dial-up networking device.

Syntax

`[form].object.BytesOut`

Remarks

The **BytesOut** property returns the number of bytes that have been transmitted by the dial-up networking device. If the control is unable to determine the number of bytes transmitted, it will return a value of zero.

This property is only supported with applications running under Windows 98 and Windows 2000. A general purpose application designed to run on all of the common Windows platforms should expect that this property may return zero as a value.

Data Type

Integer (Int32)

See Also

[BytesIn Property](#), [ConnectSpeed Property](#)

Callback Property

Specifies that the server should call the system back.

Syntax

[*form*.]*object*.**Callback** [= { True | False }]

Remarks

Setting the **Callback** property specifies that the server should call the user back at the telephone number specified by the **CallbackNumber** property. This property is ignored unless the user has "Set By Caller" callback permission on the server.

Data Type

Boolean

See Also

[CallbackNumber Property](#), [PhoneEntry Property](#), [PhoneNumber Property](#)

CallbackNumber Property

Specifies the telephone number for the server to call back on.

Syntax

`[form.]object.CallbackNumber [ = number ]`

Remarks

Setting the **CallbackNumber** property specifies that the server should call the user back at the given telephone number. This property is ignored unless the user has "Set By Caller" callback permission on the server. Assigning an asterisk to this property causes the number stored in the phone book entry to be used for callback.

Data Type

String

See Also

[Callback Property](#), [PhoneEntry Property](#), [PhoneNumber Property](#)

Connection Property

Return the handle for the specified dial-up networking session.

Syntax

`[form.]object.Connection(Index)`

Remarks

The **Connection** property array can be used to enumerate the active dial-up networking sessions on the local system. The index is zero-based, and the number of connections is returned by the **Connections** property. The property returns a long integer value which represents the handle to the session. Setting the **Handle** property to this value will cause the control to inherit the session and the control's properties will be updated with information about the connection.

Specifying an index greater than the number of available connections will generate an error.

Data Type

Integer (Int32)

See Also

[AutoConnect Property](#), [Connections Property](#), [IsConnected Property](#)

Connections Property

Return the number of active dial-up networking sessions.

Syntax

`[form.]object.Connections`

Remarks

The **Connections** property returns the number of active dial-up networking connections on the local system. A value of zero indicates that there is no dial-up networking connection. This property is used in conjunction with the **Connection** property array to enumerate the connections on the current system.

Data Type

Integer (Int32)

See Also

[AutoConnect Property](#), [Connection Property](#), [IsConnected Property](#)

ConnectSpeed Property

Returns the line speed for the current dial-up networking connection.

Syntax

`[form].object.ConnectSpeed`

Remarks

The **ConnectSpeed** property returns the speed, in bytes per second, at which the current dial-up networking device has established a connection. If the control is unable to determine the connection speed, it will return a value of zero.

This property is only supported with applications running under Windows 98, Windows NT 4.0 and later versions. A general purpose application designed to run on all of the common Windows platforms should expect that this property may return zero as a value.

Data Type

Integer (Int32)

See Also

[BytesIn Property](#), [BytesOut Property](#)

CountryCode Property

Gets and sets the country code for the current phonebook entry.

Syntax

`[form].object.CountryCode [= code ]`

Remarks

The **CountryCode** property specifies the numeric country code for the current phonebook entry. If this value is zero, then the country and area code information is not used when dialing the phone number. The country code for the United States is 1.

Data Type

Integer (Int32)

See Also

[AreaCode Property](#), [CountryName Property](#), [LoadEntry Method](#), [SaveEntry Method](#)

CountryName Property

Gets and sets the country name for the current phonebook entry.

Syntax

`[form].object.CountryName [= country ]`

Remarks

The **CountryName** property returns the name of the country associated with the country code used when dialing the current phonebook entry. If no country code has been specified, this property will return an empty string. Setting this property to the name of a country will change the current country code. If no area code has been defined, and the country code specifies the current dialing location, the **AreaCode** property will be updated to the current area code.

Data Type

String

See Also

[AreaCode Property](#), [CountryCode Property](#), [LoadEntry Method](#), [SaveEntry Method](#)

DefaultGateway Property

Set the default route for IP packets through the dial-up adapter.

Syntax

[*form*].**object.DefaultGateway** [= { True | False }]

Remarks

The **DefaultGateway** property is used to determine the default route for IP packets. If set to True, then packets are routed through the dial-up networking adapter when the connection is active. The value of this property corresponds to the "Use Default Gateway" checkbox on the TCP/IP configuration dialog.

Data Type

Boolean

DeviceCount Property

Returns the number of dial-up networking devices available.

Syntax

`[form].object.DeviceCount`

Remarks

The **DeviceCount** property returns the number of dial-up networking devices available. This property can be used in conjunction with the **DeviceEntry** property array to enumerate the devices.

Data Type

Integer (Int32)

See Also

[DeviceEntry Property](#), [DeviceName Property](#), [DeviceType Property](#)

DeviceEntry Property

Return the name of the specified device entry.

Syntax

`[form].object.DeviceEntry( Index )`

Remarks

The **DeviceEntry** property array can be used in conjunction with the **DeviceCount** property to enumerate the available dial-up networking devices. Typically this is used to provide a user with a selection of dial-up devices. The device used by the current phonebook entry can be changed by setting the **DeviceName** property to one of the device entry values.

Note that you should first set the **DeviceType** property to the type of device which you wish to enumerate. The default device type is "modem", for serial analog modems or other devices which recognize the AT command set.

Data Type

String

See Also

[DeviceCount Property](#), [DeviceName Property](#), [DeviceType Property](#)

DeviceName Property

Gets and sets the device name for the current dial-up networking connection.

Syntax

[*form*.]*object*.**DeviceName** [= *devicename*]

Remarks

The **DeviceName** property returns a description of the device that the connection was established on. For example, the string "US Robotics Sportster 28000" may be returned for a modem. Note that this property value may change if the **DeviceType** property is modified. Setting this property will change the device used to establish the dial-up networking connection. Changes to this property value should be made after changes to the **DeviceType** property.

To enumerate a list of available devices for a given device type, use the **DeviceCount** property and **DeviceEntry** property array.

Data Type

String

See Also

[DeviceCount Property](#), [DeviceEntry Property](#), [DeviceType Property](#)

DeviceType Property

Gets and sets the device type for the current dial-up networking connection.

Syntax

[*form*.]**object.DeviceType** [= *devicetype*]

Remarks

The **DeviceType** property returns the type of device that the connection was established with. Setting this property will change the type of device that will be used to establish the connection. Valid device names are:

Value	Description	
rasDeviceModem	modem	An internal or external serial analog modem device, or other serial communications device which supports the AT command set
rasDeviceISDN	isdn	An ISDN terminal adapter. Note that some ISDN devices, such as the 3Com ImpactIQ are considered modem devices.
rasDeviceX25	x25	An X25 device adapter.
rasDeviceVPN	vpn	A virtual private network connection.
RasDevicePad	pad	A packet assembler/disassembler.

Because changing the device type can change the current device name, it is recommended that applications change this property value before changing the value of the **DeviceName** property.

Data Type

String

See Also

[DeviceCount Property](#), [DeviceEntry Property](#), [DeviceName Property](#)

DynamicAddress Property

Configure the current phonebook entry to use a dynamic IP address.

Syntax

`[form].object.DynamicAddress [= { True | False } ]`

Remarks

The **DynamicAddress** property determines if the current phonebook entry should use a dynamically assigned IP address. If this property is set to True, then an IP address is assigned to the dial-up adapter when the connection is established. If set to False, then the dial-up adapter IP address is set to the value of the **InternetAddress** property.

Data Type

Boolean

See Also

[DynamicNameServers Property](#), [InternetAddress Property](#)

DynamicNameServers Property

Configure the current phonebook entry to use dynamic nameservers.

Syntax

[*form*].*object*.DynamicNameServers [= { True | False }]

Remarks

The **DynamicNameServers** property determines if the current phonebook entry should use dynamically assigned nameservers. If this property is set to True, then one or more nameservers are assigned to the dial-up adapter when the connection is established. If set to False, then the dial-up adapter nameservers are set to the values specified by the **NameServer** property array.

Data Type

Boolean

See Also

[DynamicAddress Property](#), [InternetAddress Property](#), [NameServer Property](#)

FramingProtocol Property

Gets and sets the framing protocol for the current phonebook entry.

Syntax

[*form*].**object.FramingProtocol** [= *protocol*]

Remarks

The **FramingProtocol** property is used to set or return the framing protocol used for the current phonebook entry. The following values may be specified:

Value	Description
rasFramingProtocolPpp	Point-to-Point Protocol (PPP). This is the most common protocol used by Internet Service Providers (ISPs).
rasFramingProtocolSlip	Serial Line Internet Protocol (SLIP). This is a protocol commonly used when connecting to older UNIX systems.
rasFramingProtocolRas	A proprietary Microsoft protocol implemented in Windows for Workgroups 3.11 and Windows NT 3.1

Note that unless there is a specific need for the application to use SLIP or the Microsoft protocol, it is recommended that PPP always be selected as the framing protocol.

Data Type

Integer (Int32)

See Also

[NetworkProtocol Property](#)

Handle Property

Gets and sets the handle for the current dial-up networking connection.

Syntax

object.Handle [= *hrasconn*]

Remarks

The **Handle** property returns the handle to the current dial-up networking connection, or a value of zero if the control has not been used to establish a connection. Setting the value of this property to a valid handle causes the control to inherit the specified connection, and the control's properties will be updated with information about that connection. This enables an application to monitor and control a connection that was established by the user or another program.

Setting the **Handle** property to a value of zero causes the control to release the current connection, however it will not cause the dial-up networking session to terminate. To disconnect from the server, the **Disconnect** method must be called by the application. Setting the property to a non-zero value which does not specify a valid handle will generate an error.

Data Type

Integer (Int32)

See Also

[AutoConnect Property](#), [Connection Property](#), [Connections Property](#), [IsConnected Property](#)

InternetAddress Property

Return the IP address assigned to the current dial-up networking session.

Syntax

object.InternetAddress [= *ipaddress*]

Remarks

The **InternetAddress** property returns the IP address assigned to the current dial-up networking session. If no connection has been established, or the connection has not been made with a PPP server, then this property will return an empty string. If the **DynamicAddress** property is set to False, changing this property value will update the IP address assigned to the current phonebook entry.

The IP address may only be changed before a connection is established.

Data Type

String

See Also

[DynamicAddress Property](#), [ServerAddress Property](#)

Interval Property

Gets and sets the interval at which the connection is monitored.

Syntax

`[form.]object.Interval` [= *milliseconds*]

Remarks

The **Interval** property specifies the interval, in milliseconds, at which the connection is monitored by the control. The minimum value of 0 indicates that the control should not monitor the connection. The maximum interval value is 65536 milliseconds, which is slightly more than one minute. The default value is 1000, which causes the control to check the connection status every second.

Note that setting the property value to zero will prevent the control from detecting certain conditions, such as a disconnected telephone line or a modem that is turned off.

Data Type

Integer (Int32)

See Also

[OnStatus Event](#), [OnTimeout Event](#), [Timeout Property](#)

IpHeaderCompression Property

Configure the current phonebook entry to enable IP header compression.

Syntax

[*form*].*object*.DynamicAddress [= { True | False }]

Remarks

The **IpHeaderCompression** property is used to enable or disable IP header compression. If set to True, when a connection is established, RAS will negotiate with the dial-up server to use header compression. If set to False, header compression will not be negotiated. This property corresponds to the "Use IP Header Compression" checkbox on the TCP/IP configuration dialog.

Data Type

Boolean

See Also

[DynamicNameServers Property](#), [InternetAddress Property](#), [SoftwareCompression Property](#)

IsConnected Property

Determine if the control is connected to a server.

Syntax

object.IsConnected

Remarks

The read-only **IsConnected** property is used to determine if the control has connected to the server. A value of true indicates that the connection has been established.

Note that the **IsConnected** property should not be used to determine if an active dial-up networking connection has been established by another application. The property will only return True if the control has been used to establish the connection itself, or if a connection is inherited by setting either the **AutoConnect** or **Handle** properties. To determine if there are any active dial-up networking connections, check the value of the **Connections** property.

Data Type

Boolean

See Also

[AutoConnect Property](#), [AutoDisconnect Property](#), [Connection Property](#), [Connections Property](#), [PhoneEntry Property](#), [Connect Method](#), [Disconnect Method](#)

IsInitialized Property

Determine if the control has been initialized.

Syntax

object.IsInitialized

Remarks

The **IsInitialized** property is used to determine if the current instance of the control has been initialized properly. Normally this is done automatically when the control is loaded, however there are circumstances where the control may not be able to initialize itself. If this property returns **False**, the application must call the **Initialize** method to initialize the control before performing any other operation.

The most common reason that the control may not initialize correctly is that no valid development or runtime license key can be found or the license key that was provided is invalid. It may also indicate a problem with the system configuration or user access rights, such as not being able to load the required networking libraries or not being able to access the system registry.

Data Type

Boolean

See Also

[Initialize Method](#)

LastError Property

Gets and sets the last error that occurred on the control.

Syntax

object.**LastError** [= *value*]

Remarks

The **LastError** property can be read to determine the last error that occurred for this control. If a value is assigned to this property, it must either be zero (to clear the error) or a valid error code for the control.

Data Type

Integer (Int32)

See Also

[LastErrorString Property](#), [ThrowError Property](#), [OnError Event](#)

LastErrorString Property

Return a description of the last error that occurred.

Syntax

object.LastErrorString

Remarks

The **LastErrorString** property returns a string that contains a description of the last error that occurred.

Data Type

String

See Also

[LastError Property](#), [ThrowError Property](#), [OnError Event](#)

LcpExtensions Property

Configure the current phonebook entry to use PPP LCP extensions.

Syntax

`[form].object.LcpExtensions [= { True | False } ]`

Remarks

The **LcpExtensions** property determines if the PPP LCP extensions defined in RFC 1570 will be used. If the PPP framing protocol is being used for the dial-up connection, it is recommended that this property be set to True. However, some older implementations of PPP may require that this property be set to False in order to establish a connection.

Data Type

Boolean

See Also

[FramingProtocol Property](#)

LocalNumber Property

Gets and sets the local phone number specified in the phonebook entry.

Syntax

`[form].object.LocalNumber [= number ]`

Remarks

The **LocalNumber** property sets or returns the local phone number that is specified in the current phonebook entry. If the **CountryCode** property has a value of zero, then the local number is dialed to connect to the server. If the **CountryCode** property is set to a valid country code, then RAS will also use the country and area code values when dialing the phone number.

Note that this property only determines the local phone number for the phonebook entry, and can be overridden by setting the **PhoneNumber** property to a specific value.

Data Type

String

See Also

[AreaCode Property](#), [CountryCode Property](#), [CountryName Property](#), [PhoneNumber Property](#)

ModemLights Property

Enable or disable the dial-up networking system tray icon.

Syntax

`[form].object.ModemLights [= { True | False } ]`

Remarks

The **ModemLights** property determines if the dial-up networking icon in the system tray is displayed when a connection is established.

Data Type

Boolean

See Also

[ModemSpeaker Property](#)

ModemSpeaker Property

Enable or disable the modem speaker.

Syntax

`[form].object.ModemSpeaker [= { True | False } ]`

Remarks

The **ModemSpeaker** property determines if the modem speaker is enabled when dialing the server. If the property is set to False, the modem will be silent when dialing the telephone number and establishing the connection. Note that setting this property to True will not force the speaker on if the modem hardware has been configured to explicitly disable the speaker.

To disable the speaker, the modem must support changes to the speaker volume. Disabling the speaker is typically done by instructing the modem to set the speaker volume to zero.

Data Type

Boolean

See Also

[ModemLights Property](#)

NameServer Property

Gets and sets the IP addresses of the nameservers assigned to the current phonebook entry.

Syntax

`[form].object.NameServer( Index ) [= ipaddress ]`

Remarks

The **NameServer** property array is used to set or return the nameserver IP addresses assigned to the current phonebook entry. The index value may range from 0 to 3:

Index	Description
0	Primary DNS nameserver IP address
1	Alternate DNS nameserver IP address
2	Primary WINS nameserver IP address
3	Alternate WINS nameserver IP address

Setting the property array to an IP address changes the corresponding address assigned to the phonebook entry. Note that assigned nameserver addresses are only used if the **DynamicNameServers** property has been set to False. If dynamic nameservers are assigned to the session, this property array will not return those addresses, it will return empty strings.

Data Type

String

See Also

[DynamicNameServers Property](#)

NetworkLogon Property

Configure the current phonebook entry to logon to the network.

Syntax

[*form*].*object*.**NetworkLogon** [= { True | False }]

Remarks

The **NetworkLogon** property determines if RAS automatically logs on to the network after a connection has been established. This property currently has no effect under Windows NT.

Data Type

Boolean

See Also

[DynamicNameServers Property](#), [InternetAddress Property](#)

NetworkProtocol Property

Gets and sets the network protocol for the current phonebook entry.

Syntax

[*form*].**object.NetworkProtocol** [= *protocol*]

Remarks

The **NetworkProtocol** property is used to set or return the network protocol used for the current phonebook entry. The following values may be specified:

Value	Description
rasNetworkProtocolNetBEUI	Negotiate the NetBEUI protocol.
rasNetworkProtocolIpx	Negotiate the IPX protocol.
rasNetworkProtocolIp	Negotiate the TCP/IP protocol.

These values may be combined if multiple protocols should be negotiated when the connection is established. Note that unless there is a specific need for the application to use the NetBEUI or IPX protocols, it is recommended that only the TCP/IP protocol be specified.

Data Type

Integer (Int32)

See Also

[FramingProtocol Property](#)

Password Property

The password required to establish a connection with the server.

Syntax

object.Password [= *password*]

Remarks

The **Password** property specifies the password required to establish a connection with the server. Note that this may not be the same password that is used to login to the server using terminal emulation software.

Data Type

String

See Also

[UserName Property](#), [UserDomain Property](#)

PhoneBook Property

Sets the file name of the Remote Access phone book to use.

Syntax

`[form.]object.PhoneBook [= filename ]`

Remarks

The **PhoneBook** property specifies the file name of the Remote Access phone book. Setting this property to an empty string causes the default phone book to be used.

Data Type

String

See Also

[PhoneBookEntry Property](#), [PhoneBookEntries Property](#), [UserPhoneBook Property](#)

PhoneBookEntries Property

Return the number of entries in the current phone book.

Syntax

`[form.]object.PhoneBookEntries`

Remarks

The **PhoneBookEntries** property returns the number of entries in the current phone book. A value of zero indicates that no phone book entries are available.

Data Type

Integer (Int32)

See Also

[PhoneBookEntry Property](#), [PhoneEntry Property](#)

PhoneBookEntry Property

Return the name for the specified phone book entry.

Syntax

`[form.]object.PhoneBookEntry(Index)`

Remarks

The **PhoneBookEntry** property array contains a list of the entries in the current phone book, and may be used to establish a connection with a server. Specifying an index greater than the number of available entries in the phone book will generate an error.

Data Type

String

See Also

[PhoneBookEntries Property](#), [PhoneEntry Property](#)

PhoneEntry Property

Specify the phone book entry to use to establish a connection with a server.

Syntax

[*form.*]*object*.**PhoneEntry** [= *entry*]

Remarks

The **PhoneEntry** property can be used to specify a phone book entry to use to connect with a server. The entry name identifies a communications profile which includes the telephone number, callback number and domain name of the server. Setting the **PhoneEntry** property to an empty string indicates that a telephone number will be provided to establish the connection.

Data Type

String

See Also

[PhoneBookEntries Property](#), [PhoneBookEntry Property](#), [PhoneNumber Property](#), [LoadEntry Method](#)

PhoneNumber Property

Specifies the telephone number of the server.

Syntax

`[form.]object.PhoneNumber [= value ]`

Remarks

The **PhoneNumber** property specifies the telephone number of the server. If this property is not set, then the **PhoneEntry** property must be set to a valid phone book entry. If both the **PhoneNumber** and **PhoneEntry** properties are defined, the **PhoneNumber** property will override the value specified in the phone book.

Data Type

String

See Also

[PhoneEntry Property](#), [CallbackNumber Property](#)

RequireEncryption Property

Configure the current phonebook entry to require secure authentication.

Syntax

`[form].object.RequireEncryption [= { True | False } ]`

Remarks

The **RequireEncryption** property determines if encryption is required during PPP authentication. If the property is set to True, then only secure password schemes can be used to authenticate the client. If the property is set to False, the client can use the PAP plain-text authentication protocol to authenticate the client. Some older PPP implementations may require that this property be set to False in order to establish a connection.

Data Type

Boolean

ScriptFile Property

Gets and sets the name of the script file for the current phonebook entry.

Syntax

`[form].object.ScriptFile [= filename ]`

Remarks

The **ScriptFile** property specifies the name of the login script used to establish a connection with the server. This property must be set to the full pathname of the script file. If a script file is not required, then this property should be set to an empty string.

Data Type

String

See Also

[Terminal Property](#)

ServerAddress Property

Return the IP address of the dial-up networking server.

Syntax

[*form.*]object.**ServerAddress**

Remarks

The **ServerAddress** property returns the IP address of the dial-up networking server that the local host has connected to. If no connection has been established, or the connection has not been made with a PPP server, then this property will return an empty string. This property may also return an empty string if the server did not provide this information during the connection process.

Data Type

String

See Also

[InternetAddress Property](#)

SoftwareCompression Property

Configure the current phonebook entry to negotiate software compression.

Syntax

`[form].object.SoftwareCompression [= { True | False } ] ]`

Remarks

The **SoftwareCompression** property determines if data compression is negotiated during the connection. If the property is set to True, then the client will negotiate a compatible compression protocol. Software compression should only be disabled if the client is unable to establish a connection with the server.

Data Type

Boolean

See Also

[IpHeaderCompression Property](#)

Status Property

Return the current status of the control.

Syntax

object.Status

Remarks

This read-only property returns the status of the control. It may be one of the following values:

Value	Description	
-1	rasStatusUnused	No connection has been established
rasStatusOpenPort	The communications port is about to be opened	
rasStatusPortOpened	The communications port has been opened	
rasStatusConnectDevice	A device is about to be connected	
rasStatusDeviceConnected	A device has been connected successfully	
rasStatusAllDevicesConnected	All devices have been connected	
rasStatusAuthenticate	Authenticating username and password	
rasStatusAuthNotify	An authentication event has occurred	
rasStatusAuthRetry	Requesting authentication with new credentials	
rasStatusAuthCallback	The server has requested a callback number	
rasStatusAuthChangePassword	The user has requested to change the password	
rasStatusAuthProject	Registering computer on the network	
rasStatusAuthLinkSpeed	The link speed calculation phase is starting	
rasStatusAuthAck	An authentication request is being acknowledged	
rasStatusReAuthenticate	Authenticating username and password	
rasStatusAuthenticated	The user has been authenticated	
rasStatusPrepareForCallback	The line is about to be disconnected in preparation for callback	
rasStatusWaitForModemReset	The modem is resetting itself in preparation for callback	
rasStatusWaitForCallback	Waiting for callback from server	

rasStatusProjected	Protocol specific information has been negotiated
rasStatusStartAuthentication	User authentication is being initiated
rasStatusCallbackComplete	Callback completed and resuming authentication
rasStatusLogonNetwork	Logging on to the network
rasStatusSubEntryConnected	A subentry has been connected
rasStatusSubEntryDisconnected	A subentry has been disconnected
rasStatusInteractive	Initiating interactive login session
rasStatusRetryAuthentication	Retrying user authentication
rasStatusCallbackSetByCaller	Callback has been set by caller
rasStatusPasswordExpired	Password has expired
rasStatusConnected	Connected to server
rasStatusDisconnected	Disconnected from server

Data Type

Integer (Int32)

See Also

[AutoConnect Property](#), [AutoDisconnect Property](#), [Interval Property](#), [IsConnected Property](#), [OnStatus Event](#)

Terminal Property

Determine if a terminal window is displayed during the connection process.

Syntax

[*form*].*object*.Terminal [= *value*]

Remarks

The **Terminal** property array is used to control if a terminal window is displayed during the dial-up networking connection process. The property may be set to one of the following values:

Value	Description
No terminal window is displayed	
Terminal window is displayed before dialing	
Terminal window is displayed after dialing. Do not use if scripting has been enabled.	
Terminal window is display before and after dialing. Do not use if scripting has been enabled.	

The terminal window can be used to allow user input before and/or after the dial-up networking connection has been established. If scripting has been enabled by setting the **ScriptFile** property, no terminal window should be displayed after the connection. This is because scripting has it's own terminal implementation.

Note that this property is only supported on Windows NT 4.0 and later versions of the operating system. Displaying a terminal window also imposes several restrictions on the behavior of the control. Because of how the Remote Access Services API is implemented by Microsoft, a connection dialog will be displayed after the **Connect** method is called if the **Terminal** property is non-zero. Setting this property to a non-zero value will also disable any asynchronous event notifications. It is not recommended that you set this property unless it is absolutely necessary.

Data Type

Integer (Int32)

See Also

[ScriptFile Property](#), [Connect Method](#)

ThrowError Property

Enables or disables error handling by the application.

Syntax

object.**ThrowError** [= { True | False }]

Remarks

Error handling for methods can be performed using either return codes or exceptions, depending on the value of this property.

If the **ThrowError** property is set to False, methods will return a status value indicating success or failure. Applications should check the return value of each method and use the **LastError** and **LastErrorString** properties to determine the cause of any error.

If the **ThrowError** property is set to True, errors occurring within the control will be raised to the host application as ActiveX exceptions. In addition, the **OnError** event will fire. This allows applications to use language-specific exception handling mechanisms, such as Visual Basic error handling, instead of checking individual return values after each method call.

Even when the **ThrowError** property is set to True, the **LastError** and **LastErrorString** properties will still be updated with information about the most recent error.

If an error occurs while accessing a property value, an exception will be raised regardless of the value of the **ThrowError** property. However, the **OnError** event will not be fired for property access errors.

Data Type

Boolean

See Also

[LastError Property](#), [LastErrorString Property](#), [OnError Event](#)

Timeout Property

Gets and sets the number of seconds until a connection attempt fails.

Syntax

`[form.]object.Timeout [= seconds ]`

Remarks

This property specifies the number of seconds that the control has to establish a connection with a server. If a connection is not established within that time period, the **OnTimeout** event is fired and the control is reset. The default value for this property is 20 seconds.

Data Type

Integer (Int32)

See Also

[Interval Property](#), [OnStatus Event](#), [OnTimeout Event](#)

UserDomain Property

Specifies the NT domain on which user authentication is to occur.

Syntax

[*form.*]object.UserDomain [= *domain*]

Remarks

The **UserDomain** property is used to specify the NT domain on which the user name and password will be authenticated. An empty string specifies the domain in which the Remote Access server is a member. An asterisk specifies the domain stored in the phone book entry.

Data Type

String

See Also

[Password Property](#), [UserName Property](#)

UserName Property

Set the user name that is required to establish a connection with the server.

Syntax

`[form.]object.UserName [= name ]`

Remarks

The **UserName** property specifies the user that is logging into the server, and is required for authentication purposes.

Data Type

String

See Also

[Password Property](#), [UserDomain Property](#)

UserPhoneBook Property

Returns the name of the default user phonebook.

Syntax

`[form].object.UserPhoneBook`

Remarks

The **UserPhoneBook** property returns the name of the default user phonebook. The value returned depends on how the user has configured dial-up networking, specifically whether the system, user or alternate phonebook has been selected. The current phonebook can be changed by setting the **PhoneBook** property.

Note that this property always returns an empty string under Windows 98 since phonebooks are not used (entries are stored in the system registry).

Data Type

String

See Also

[PhoneBook Property](#)

Version Property

Return the current version of the control.

Syntax

object.**Version**

Remarks

The **Version** property returns the current version of the object. This can be used by an application for validation purposes.

Data Type

String

Internet Dialer Control Methods

Method	Description
Connect	Establish a connection with a server
CreateEntry	Create a new entry in the current phonebook
DeleteEntry	Delete a phonebook entry from the local system
Disconnect	Terminate the connection with a server
EditEntry	Edit an existing phonebook entry on the local system
Initialize	Initialize the component and load the Remote Access Services library
LoadEntry	Load the specified entry from the current phonebook
RenameEntry	Rename an existing phonebook entry
Reset	Resets the control state and disconnects the current session
SaveEntry	Save the specified entry to the current phonebook
Uninitialize	Uninitialize the component and unload the Remote Access Services library

Connect Method

Establish a connection with a server.

Syntax

object.Connect([*EntryName*])

Parameters

EntryName

An optional string value that specifies the name of the phonebook entry to use to establish the connection. If this argument is not provided, the value of the **PhoneEntry** property is used.

Return Value

A value of zero is returned if the connection was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Connect** method establishes a dial-up networking connection with a service provider using the specified phonebook entry. If this method is called without any arguments and **PhoneEntry** property has not been set, then the current values of the **PhoneNumber**, **UserName**, **UserDomain**, and **Password** properties will be used to create a temporary phonebook entry.

See Also

[Disconnect Method](#), [CallbackNumber Property](#), [Password Property](#), [PhoneEntry Property](#), [PhoneNumber Property](#), [UserName Property](#), [UserDomain Property](#)

CreateEntry Method

Create a new entry in the current phonebook.

Syntax

object.CreateEntry

Parameters

None.

Return Value

A value of zero is returned if the operation was successful, otherwise a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CreateEntry** method displays a dialog box which allows the user to create a new phonebook entry on the system. If you do not wish to display a dialog box, use the **SaveEntry** method instead.

See Also

[DeleteEntry Method](#), [EditEntry Method](#), [RenameEntry Method](#), [SaveEntry Method](#)

DeleteEntry Method

Delete a phonebook entry from the local system.

Syntax

object.DeleteEntry([*EntryName*])

Parameters

EntryName

An optional string value which specifies specifies the phonebook entry to delete. If this argument is not provided, the value of the **PhoneEntry** property will be used.

Return Value

A value of zero is returned if the operation was successful, otherwise a non-zero error code is returned which indicates the cause of the failure.

See Also

[CreateEntry Method](#), [EditEntry Method](#), [RenameEntry Method](#)

Disconnect Method

Terminate the dial-up networking connection.

Syntax

object.Disconnect

Parameters

None.

Return Value

A value of zero is returned if the connection was terminated successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

See Also

[Connect Method](#)

EditEntry Method

Edit an existing phonebook entry on the local system.

Syntax

object.EditEntry([*EntryName*])

Parameters

EntryName

An optional string value which specifies the phonebook entry to edit. If this argument is not provided, the value of the **PhoneEntry** property will be used.

Return Value

A value of zero is returned if the operation was successful, otherwise a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **EditEntry** method edits the specified entry from the local phonebook. This will cause a dialog box to be displayed from which the user can change the connection information. If you do not want to display a dialog, then use the **SaveEntry** method instead.

See Also

[CreateEntry Method](#), [DeleteEntry Method](#), [RenameEntry Method](#), [SaveEntry Method](#)

Initialize Method

Initialize the control and validate the runtime license key.

Syntax

object.Initialize([*LicenseKey*])

Parameters

LicenseKey

An optional string value which specifies a runtime license key used to initialize the control. If the license key is omitted or passed as an empty string, a development license must be installed on the local system.

Return Value

A value of zero is returned if the control was initialized successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

This method dynamically loads other system libraries and allocates thread local storage. In most cases, it is not necessary to call this method directly because it is automatically invoked when an instance of the control is created by the container. However, if the control is created dynamically using **CreateObject** or a similar method, this must be the first method that is called before you attempt to modify any property values or invoke other methods. Failure to initialize the control may result in subsequent errors and/or cause an exception to be raised.

Each time the **Initialize** method is invoked, it increments an internal counter that keeps track of the number of times that it has been called by any thread in the current process. The **Uninitialize** method decrements this counter, and when the usage count drops to zero, the control will automatically unload the system libraries that it has dynamically loaded and will destroy the process heap that was allocated when the first instance of the control was created.

If the control is being used within another DLL, it is important that you do not attempt to create an instance of the control or call the **Initialize** method from within the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is written in C++ and it is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for any static and global C++ objects and has the same restrictions.

Example

```
Set rasDialer = CreateObject("SocketTools.Dialer.11")

nError = rasDialer.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize the SocketTools component"
End If
```

See Also

[IsInitialized Property](#), [Uninitialize Method](#)

LoadEntry Method

Load the specified entry from the current phonebook.

Syntax

object.LoadEntry([*EntryName*])

Parameters

EntryName

An optional string value which specifies the phonebook entry to load. If this argument is not provided, the current entry is reloaded from the phonebook, abandoning any changes that have been made.

Return Value

A value of zero is returned if the operation was successful, otherwise a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **LoadEntry** method loads the specified entry from the current phonebook and sets the control properties to match the configuration.

See Also

[PhoneBook Property](#), [PhoneEntry Property](#)

RenameEntry Method

Rename an existing phonebook entry.

Syntax

object.RenameEntry(*OldName*, *NewName*)

Parameters

OldName

A string value that specifies the name of the phonebook entry to be renamed.

NewName

A string value that specifies the new name of the phonebook entry. This name must not already exist for another connectoid in the current phonebook.

Return Value

A value of zero is returned if the operation was successful, otherwise a non-zero error code is returned which indicates the cause of the failure.

See Also

[CreateEntry Method](#), [DeleteEntry Method](#), [EditEntry Method](#)

Reset Method

Reset the internal state of the control.

Syntax

object.Reset

Parameters

None.

Return Value

None.

Remarks

The **Reset** method resets the internal state of the control. Property values are initialized to their internal defaults, active dial-up connections will be terminated and any handles allocated by the control will be released. Any property changes to the current phonebook entry will be ignored, reverting to their previous values.

See Also

[Initialize Method](#), [Uninitialize Method](#)

SaveEntry Method

Save the specified entry to the current phonebook.

Syntax

object.**SaveEntry**([*EntryName*])

Parameters

EntryName

An optional string value that specifies the name of phonebook entry. If this argument is not provided, the current value of the **PhoneEntry** property is used.

Return Value

A value of zero is returned if the operation was successful, otherwise a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **SaveEntry** method saves the specified entry to the current phonebook, based on the current control properties. If the entry does not exist, it will be created. If an entry by that name already exists, it will be overwritten. Note that unlike the **CreateEntry** method, this method does not display any dialogs.

See Also

[PhoneBook Property](#), [PhoneEntry Property](#), [CreateEntry Method](#), [EditEntry Method](#)

Uninitialize Method

Uninitialize the component and unload the Remote Access Services library.

Syntax

object.Uninitialize

Parameters

None.

Return Type

None.

Remarks

The **Uninitialize** method terminates any active dial-up networking connection established by the control and unloads the Remote Access Services (RAS) library. This method is not typically used because any resources that have been allocated by an instance of the control will automatically be released when it is destroyed. To prevent the connection from being terminated when the control is uninitialized, set the **AutoDisconnect** property to False or set the **Handle** property to a value of zero before calling this method.

Each time the **Initialize** method is invoked, it increments an internal counter that keeps track of the number of times that it has been called by any thread in the current process. The **Uninitialize** method decrements this counter, and when the usage count drops to zero, the control will automatically unload the system libraries that it has dynamically loaded and will destroy the process heap that was allocated when the first instance of the control was created. An application should only call the **Uninitialize** method if it has explicitly called the **Initialize** method.

See Also

[AutoDisconnect Property](#), [Handle Property](#), [Connect Method](#), [Disconnect Method](#), [Initialize Method](#)

Internet Dialer Control Events

Event	Description
OnConnect	This event occurs when a connection is established
OnDisconnect	This event occurs when a connection is terminated
OnError	This event occurs when a control error occurs
OnStatus	This event occurs when the control state changes
OnTimeout	This event occurs when the control is unable to establish a connection

OnConnect Event

The **OnConnect** event is generated when a connection is established.

Syntax

Sub *object_OnConnect* ([*Index As Integer*])

Remarks

The **OnConnect** event is generated when a successful connection has been established with the server. To monitor the progress of the connection attempt, use the **OnStatus** event.

See Also

[OnDisconnect Event](#), [OnStatus Event](#)

OnDisconnect Event

The **OnDisconnect** event is generated when a connection is terminated.

Syntax

Sub *object_OnDisconnect* ([*Index As Integer*])

Remarks

The **OnDisconnect** event is generated when the connection is terminated by the server.

See Also

[OnConnect Event](#)

OnError Event

The **OnError** event is generated when a control error occurs.

Syntax

```
Sub object_OnError ( [Index As Integer,] ByVal ErrorCode As Variant, ByVal Description As Variant )
```

Remarks

This event occurs when an error occurs during a control operation. The **OnError** event is typically fired when a method is called which results in an error, or an error occurs during the connection or authentication process.

The ***ErrorCode*** argument specifies the numeric error code. The Remote Access Services subsystem returns errors in the range of 600 to 800. These are automatically converted to 10600 through 10800 to avoid conflicts with standard error codes. For example, error 10676 corresponds to the RAS error 676, which indicates that the line is busy.

The ***Description*** argument contains a description of the error.

See Also

[LastError Property](#), [LastErrorString Property](#), [ThrowError Property](#)

OnStatus Event

The **OnStatus** event is generated when the control state changes.

Syntax

Sub *object_OnStatus* ([*Index As Integer*,] **ByVal** *State As Variant*, **ByVal** *Description As Variant*)

Remarks

This event occurs when the status of the control changes. Typically this occurs when a connection is being established with a server.

The **State** argument is a numeric code which identifies the state of the control. This is the same value as returned by the **State** property.

The **Description** argument contains a string which describes the new state. Applications may use this value to provide feedback to the user or for logging purposes.

See Also

[AutoConnect Property](#), [AutoDisconnect Property](#), [IsConnected Property](#), [Status Property](#)

OnTimeout Event

The **OnTimeout** event is generated when the control is unable to establish a connection.

Syntax

Sub *object_OnTimeout* ([*Index As Integer*])

Remarks

This event occurs when the control is unable to establish a connection with a server in the number of seconds specified by the **Timeout** property.

See Also

[Timeout Property](#)

Internet Dialer Control Errors

Value	Description
rasErrorPending	An operation is pending
rasErrorInvalidPortHandle	An invalid port handle was detected
rasErrorPortAlreadyOpen	The specified port is already open
rasErrorBufferTooSmall	The caller's buffer is too small
rasErrorWrongInfoSpecified	Incorrect information was specified
rasErrorCannotSetPortInfo	The port information cannot be set
rasErrorPortNotConnected	The specified port is not connected
rasErrorEventInvalid	An invalid event was detected
rasErrorDeviceDoesNotExist	A device was specified that does not exist
rasErrorDevicetypeDoesNotExist	A device type was specified that does not exist
rasErrorBufferInvalid	An invalid buffer was specified
rasErrorRouteNotAvailable	A route was specified that is not available
rasErrorRouteNotAllocated	A route was specified that is not allocated
rasErrorInvalidCompressionSpecified	An invalid compression was specified
rasErrorOutOfBuffers	There were insufficient buffers available
rasErrorPortNotFound	The specified port was not found
rasErrorAsyncRequestPending	An asynchronous request is pending
rasErrorAlreadyDisconnecting	The modem or other connecting device is already disconnecting
rasErrorPortNotOpen	The specified port is not open
rasErrorPortDisconnected	A connection to the remote computer could not be established
rasErrorNoEndpoints	No endpoints could be determined
rasErrorCannotOpenPhonebook	The system could not open the phone book file
rasErrorCannotLoadPhonebook	The system could not load the phone book file
rasErrorCannotFindPhonebookEntry	The system could not find the phone book entry for this connection
rasErrorCannotWritePhonebook	The system could not update the phone book file
rasErrorCorruptPhonebook	The system found invalid information in the phone book file
rasErrorCannotLoadString	A string could not be loaded
rasErrorKeyNotFound	A key could not be found
rasErrorDisconnection	The connection was terminated by the remote computer before it could be completed
rasErrorRemoteDisconnection	The connection was closed by the remote computer
rasErrorHardwareFailure	The modem or other connecting device was disconnected due to

	hardware failure
rasErrorUserDisconnection	The user disconnected the modem or other connecting device
rasErrorInvalidSize	An incorrect structure size was detected
rasErrorPortNotAvailable	The modem or other connecting device is already in use or is not configured properly
rasErrorCannotProjectClient	Your computer could not be registered on the remote network
rasErrorUnknown	There was an unknown error
rasErrorWrongDeviceAttached	The device attached to the port is not the one expected
rasErrorBadString	A string was detected that could not be converted
rasErrorRequestTimeout	The request has timed out
rasErrorCannotGetLana	No asynchronous net is available
rasErrorNetBIOSError	An error has occurred involving NetBIOS
rasErrorServerOutOfResources	The server cannot allocate NetBIOS resources needed to support the client
rasErrorNameExistsOnNet	One of your computer's NetBIOS names is already registered on the remote network
rasErrorServerGeneralNetFailure	A network adapter at the server failed
rasErrorMsgAliasNotAdded	You will not receive network message popups
rasErrorAuthInternal	There was an internal authentication error
rasErrorRestrictedLogonHours	The account is not permitted to log on at this time of day
rasErrorAcctDisabled	The account is disabled
rasErrorPasswdExpired	The password for this account has expired
rasErrorNoDialInPermission	The account does not have permission to dial in
rasErrorServerNotResponding	The remote access server is not responding
rasErrorFromDevice	The modem or other connecting device has reported an error
rasErrorUnrecognizedResponse	There was an unrecognized response from the modem or other connecting device
rasErrorMacroNotFound	A macro required by the modem or other connecting device was not found in the configuration file
rasErrorMacroNotDefined	A command or response in the configuration file refers to an undefined macro
rasErrorMessageMacroNotFound	The message macro was not found in the configuration file
rasErrorDefaultOffMacroNotFound	The configuration file contains an undefined macro
rasErrorFileCouldNotBeOpened	The configuration file could not be opened
rasErrorDevicenameTooLong	The device name in the configuration file is too long
rasErrorDevicenameNotFound	The configuration file refers to an unknown device name
rasErrorNoResponses	The configuration file contains no responses for the command

rasErrorNoCommandFound	The configuration file is missing a command
rasErrorWrongKeySpecified	There was an attempt to set a macro not listed in configuration file
rasErrorUnknownDeviceType	The configuration file refers to an unknown device type
rasErrorAllocatingMemory	The system has run out of memory
rasErrorPortNotConfigured	The modem or other connecting device is not properly configured
rasErrorDeviceNotReady	The modem or other connecting device is not functioning
rasErrorReadingIniFile	The system was unable to read the configuration file
rasErrorNoConnection	The connection was terminated
rasErrorBadUsageInIniFile	The usage parameter in the configuration file is invalid
rasErrorReadingSectionname	The system was unable to read the section name from the configuration file
rasErrorReadingDeviceType	The system was unable to read the device type from the configuration file
rasErrorReadingDeviceName	The system was unable to read the device name from the configuration file
rasErrorReadingUsage	The system was unable to read the usage from the configuration file
rasErrorReadingMaxconnectbps	The system was unable to read the maximum connection BPS rate from the configuration file
rasErrorReadingMaxcarrierbps	The system was unable to read the maximum carrier connection speed from the configuration file
rasErrorLineBusy	The phone line is busy
rasErrorVoiceAnswer	A person answered instead of a modem or other connecting device
rasErrorNoAnswer	The remote computer did not respond
rasErrorNoCarrier	The system could not detect the carrier
rasErrorNoDialtone	There was no dial tone
rasErrorInCommand	The modem or other connecting device reported a general error
rasErrorWritingSectionname	There was an error in writing the section name
rasErrorWritingDevicetype	There was an error in writing the device type
rasErrorWritingDevicename	There was an error in writing the device name
rasErrorWritingMaxconnectbps	There was an error in writing the maximum connection speed.
rasErrorWritingMaxCarrierBps	There was an error in writing the maximum carrier speed
rasErrorWritingUsage	There was an error in writing the usage
rasErrorWritingDefaultOff	There was an error in writing the default-off
rasErrorReadingDefaultOff	There was an error in reading the default-off
rasErrorEmptyIniFile	The configuration file is empty
rasErrorAuthenticationFailure	Access was denied because the username and/or password was

	invalid on the domain
rasErrorPortOrDevice	There was a hardware failure in the modem or other connecting device
rasErrorNotBinaryMacro	An internal error has occurred
rasErrorDcbNotFound	An internal error has occurred
rasErrorStateMachinesNotStarted	The state machines are not started
rasErrorStateMachinesAlreadyStarted	The state machines are already started
rasErrorPartialResponseLooping	The response looping did not complete
rasErrorUnknownResponseKey	A response keyname in the configuration file is not in the expected format
rasErrorRecvBufFull	The modem or other connecting device response caused a buffer overflow
rasErrorCmdTooLong	The expanded command in the configuration file is too long
rasErrorUnsupportedBps	The modem moved to a connection speed not supported by the COM driver
rasErrorUnexpectedResponse	Device response received when none expected
rasErrorInteractiveMode	The connection needs information from you, but the application does not allow user interaction
rasErrorBadCallbackNumber	The callback number is invalid
rasErrorInvalidAuthState	The authorization state is invalid
rasErrorWritingInitbps	An internal error has occurred
rasErrorX25Diagnostic	There was an error related to the X.25 protocol
rasErrorAcctExpired	The account has expired
rasErrorChangingPassword	There was an error changing the password on the domain
rasErrorOverrun	Serial overrun errors were detected while communicating with the modem
rasErrorRasmanCannotInitialize	A configuration error on this computer is preventing this connection
rasErrorBiplexPortNotAvailable	The two-way port is initializing, wait a few seconds and redial
rasErrorNoActiveIsdnLines	No active ISDN lines are available
rasErrorNoIsdnChannelsAvailable	No ISDN channels are available to make the call
rasErrorTooManyLineErrors	Too many errors occurred because of poor phone line quality
rasErrorIpConfiguration	The Remote Access Service IP configuration is unusable
rasErrorNoIpAddresses	No IP addresses are available in the static pool of Remote Access Service IP addresses
rasErrorPppTimeout	The connection was terminated because the remote computer did not respond in a timely manner
rasErrorPppRemoteTerminated	The connection was terminated by the remote computer

rasErrorPppNoProtocolsConfigured	A connection to the remote computer could not be established
rasErrorPppNoResponse	The remote computer did not respond
rasErrorPppInvalidPacket	Invalid data was received from the remote computer
rasErrorPhoneNumberTooLong	The phone number, including prefix and suffix, is too long
rasErrorIpxcpNoDialoutConfigured	The IPX protocol cannot dial out on the modem because this computer is not configured for dialing out
rasErrorIpxcpNoDialinConfigured	The IPX protocol cannot dial in on the modem because this computer is not configured for dialing in
rasErrorIpxcpDialoutAlreadyActive	The IPX protocol cannot be used for dialing out on more than one modem
rasErrorAccessingTcpcfgDll	Cannot access TCPCFG.DLL
rasErrorNoIpRasAdapter	The system cannot find an IP adapter
rasErrorSlipRequiresIp	SLIP cannot be used unless the IP protocol is installed
rasErrorProjectionNotComplete	Computer registration is not complete
rasErrorProtocolNotConfigured	The protocol is not configured
rasErrorPppNotConverging	Your computer and the remote computer could not agree on PPP control protocols
rasErrorPppCpRejected	A connection to the remote computer could not be completed
rasErrorPppLcpTerminated	The PPP link control protocol was terminated
rasErrorPppRequiredAddressRejected	The requested address was rejected by the server
rasErrorPppNcpTerminated	The remote computer terminated the control protocol
rasErrorPppLoopbackDetected	Loopback was detected
rasErrorPppNoAddressAssigned	The server did not assign an address
rasErrorCannotUseLogonCredentials	The authentication protocol required by the server cannot use the stored password
rasErrorTapiConfiguration	An invalid dialing rule was detected
rasErrorNoLocalEncryption	The local computer does not support the required data encryption type
rasErrorNoRemoteEncryption	The remote computer does not support the required data encryption type
rasErrorRemoteRequiresEncryption	The remote computer requires data encryption
rasErrorIpxcpNetNumberConflict	The system cannot use the IPX network number assigned by the remote computer
rasErrorInvalidSmm	An internal error has occurred
rasErrorSmmUninitialized	An internal error has occurred
rasErrorNoMacForPort	An internal error has occurred
rasErrorSmmTimeout	An internal error has occurred

rasErrorBadPhoneNumber	An invalid telephone number has been specified
rasErrorWrongModule	An internal error has occurred
rasErrorInvalidCallbackNumber	The callback number contains an invalid character
rasErrorScriptSyntax	A syntax error was encountered while processing a script
rasErrorHangupFailed	The connection could not be disconnected because it was created by the multi-protocol router
rasErrorBundleNotFound	The system could not find the multi-link bundle
rasErrorCannotDoCustomdial	The system cannot perform automated dial because this connection has a custom dialer specified
rasErrorDialAlreadyInProgress	This connection is already being dialed
rasErrorRasautoCannotInitialize	Remote Access Services could not be started automatically
rasErrorConnectionAlreadyShared	Internet Connection Sharing is already enabled on the connection
rasErrorSharingChangeFailed	An error occurred while the existing Internet Connection Sharing settings were being changed
rasErrorSharingRouterInstall	An error occurred while routing capabilities were being enabled
rasErrorShareConnectionFailed	An error occurred while Internet Connection Sharing was being enabled for the connection
rasErrorSharingPrivateInstall	An error occurred while the local network was being configured for sharing
rasErrorCannotShareConnection	Internet Connection Sharing cannot be enabled
rasErrorNoSmartCardReader	No smart card reader is installed
rasErrorSharingAddressExists	Internet Connection Sharing cannot be enabled
rasErrorNoCertificate	A certificate could not be found
rasErrorSharingMultipleAddresses	Internet Connection Sharing cannot be enabled
rasErrorFailedToEncrypt	The connection attempt failed because of failure to encrypt data
rasErrorBadAddressSpecified	The specified destination is not reachable
rasErrorConnectionReject	The remote computer rejected the connection attempt
rasErrorCongestion	The connection attempt failed because the network is busy
rasErrorIncompatible	The remote computer's network hardware is incompatible with the type of call requested
rasErrorNumberChanged	The connection attempt failed because the destination number has changed
rasErrorTempfailure	The connection attempt failed because of a temporary failure
rasErrorBlocked	The call was blocked by the remote computer
rasErrorDonotdisturb	The call could not be connected because the remote computer has invoked the Do Not Disturb feature
rasErrorOutOfOrder	The connection attempt failed because the modem on the remote

	computer is out of order
rasErrorUnableToAuthenticateServer	It was not possible to verify the identity of the server
rasErrorSmartCardRequired	To dial out using this connection you must use a smart card
rasErrorInvalidFunctionForEntry	An attempted function is not valid for this connection
rasErrorCertForEncryptionNotFound	The connection requires a certificate, and no valid certificate was found
rasErrorSharingRrasConflict	Network Address Translation must be removed before enabling Internet Connection Sharing
rasErrorSharingNoPrivateLan	Internet Connection Sharing cannot be enabled
rasErrorNoDiffUserAtLogon	You cannot dial using this connection at logon time
rasErrorNoRegCertAtLogon	You cannot dial using this connection at logon time
rasErrorOakleyNoCert	The L2TP connection attempt failed because there is no valid machine certificate on your computer for security authentication
rasErrorOakleyAuthFail	The L2TP connection attempt failed because the security layer could not authenticate the remote computer
rasErrorOakleyAttribFail	The L2TP connection attempt failed because the security layer could not negotiate compatible parameters with the remote computer
rasErrorOakleyGeneralProcessing	The L2TP connection attempt failed because the security layer encountered a processing error during initial negotiations with the remote computer
rasErrorOakleyNoPeerCert	The L2TP connection attempt failed because certificate validation on the remote computer failed
rasErrorOakleyNoPolicy	The L2TP connection attempt failed because security policy for the connection was not found
rasErrorOakleyTimedOut	The L2TP connection attempt failed because security negotiation timed out
rasErrorOakleyError	The L2TP connection attempt failed because an error occurred while negotiating security
rasErrorUnknownFramedProtocol	The Framed Protocol RADIUS attribute for this user is not PPP
rasErrorWrongTunnelType	The Tunnel Type RADIUS attribute for this user is not correct
rasErrorUnknownServiceType	The Service Type RADIUS attribute for this user is neither Framed nor Callback Framed
rasErrorConnectingDeviceNotFound	A connection to the remote computer could not be established because the modem was not found or was busy
rasErrorNoEaptlsCertificate	A certificate could not be found that can be used with this Extensible Authentication Protocol
rasErrorSharingHostAddressConflict	Internet Connection Sharing cannot be enabled
rasErrorAutomaticVpnFailed	Unable to establish the VPN connection
rasErrorValidatingServerCert	Unable to verify the digital certificate sent by the server

Internet Server Control

A general purpose TCP/IP networking component for developing server applications.

Reference

- [Properties](#)
- [Methods](#)
- [Events](#)
- [Error Codes](#)

Control Information

Object Name	InternetServerCtl.InternetServer
File Name	CSWSVX12.OCX
Version	12.0.3024.1286
ProgID	SocketTools.InternetServer.12
ClassID	C1F0B98E-583F-45B3-9BA9-A8D6F45103DF
Threading Model	Apartment
Help File	CSW12HLP.CHM
Dependencies	None
Standards	RFC 768, RFC 791, RFC 793

Overview

The Internet Server ActiveX control provides a simplified interface for creating event-driven, multithreaded server applications using the TCP/IP protocol. The control interface is similar to the SocketWrench ActiveX control, however it is designed specifically to make it easier to implement a server application without requiring the need to manage multiple socket controls. In addition, the Internet Server control supports secure communications using the Transport Layer Security (TLS) protocol.

Each instance of the Internet Server control represents a server, and each active client connection is managed internally and referenced by an integer value which uniquely identifies the client session. All interaction with the server and the clients connected to it uses an event-driven model, with the program written to respond to events such as OnConnect, OnRead and OnWrite.

Developers who have used the SocketWrench ActiveX control will find the Internet Server control has a familiar interface, with a subset of properties and methods that are specific to creating a server application. Each of the network events have an extra parameter which specifies the socket handle which should be used when communicating with the client. This enables the application to communicate with multiple clients without having to create multiple socket objects or use a control array.

Requirements

The SocketTools ActiveX Edition components are self-registering controls that are compatible with any programming language that supports the Component Object Model (COM) and the ActiveX control specification. If you are using Visual Basic 6.0, Service Pack 6 (SP6) is required. We recommend

installing all available updates for your development tools. Although the Visual Basic 6.0 runtime is supported on Windows 11, the IDE itself is no longer officially supported by Microsoft. Legacy IDEs and compilers may also require elevated (administrator) privileges to run properly.

The ActiveX controls are supported on Windows 7, Windows Server 2008 R2, and later versions of both desktop and server editions of Windows. For Windows 7, Service Pack 1 (SP1) is the minimum requirement. For Windows 8 systems, the 8.1 update must be installed. On ARM-based platforms, the minimum supported version is Windows 10 build 1709.

SocketTools includes components for both 32-bit (x86) and 64-bit (x64) Windows platforms. All modern versions of Windows are 64-bit, but we continue to include 32-bit components for compatibility with legacy development tools such as Visual Basic 6.0. We also provide native ARM64 ActiveX controls for systems running Windows on ARM. Note that 32-bit applications on ARM run using x86 emulation, and Microsoft no longer supports native ARM32 applications.

If you are using VBScript, Visual Basic for Applications (VBA), or another scripting environment on an ARM-based system, be aware that 64-bit hosts will require ARM64 components, while 32-bit hosts will use x86 components under emulation. This includes applications created with Visual Basic 6.0. To reduce deployment complexity and avoid confusion, SocketTools does not include ARM32 ActiveX controls in its standard redistribution packages but they are available on request for legacy or specialized scenarios.

Distribution

When you distribute an application that uses this control, you can either install the file in the same folder as your application executable or as a shared component in the appropriate system folder. If you install the control in the system folder, it is important that you distribute the correct version for the target platform and it must be registered. If you install the control in the same folder as your executable, it is recommended that you use registration-free activation or COM redirection to ensure that the correct version of the control is loaded by the application.

Internet Server Control Properties

Property	Description
AdapterAddress	Returns the IP address associated with the specified network adapter
AdapterCount	Returns the number of available local and remote network adapters
Backlog	Gets and sets the number of client connections that may be queued by the server
ByteOrder	Gets and sets the byte order in which integer data will be written to and read from the socket
CertificateName	Gets and sets the common name for the server certificate
CertificatePassword	Gets and sets the password associated with the server certificate
CertificateStore	Gets and sets the name of the server certificate store or file
CertificateUser	Gets and sets the user that owns the server certificate
ClientAddress	Return the address of the current client session
ClientCount	Return the number of active client sessions connected to the server
ClientHandle	Return the socket handle associated with a specific client session
ClientHost	Return the hostname for the current client session
ClientId	Return a unique identifier for the current client session
ClientName	Gets and sets a unique string moniker that is associated with the current client session
ClientPort	Return the port number used by the current client session
ClientThread	Return the thread ID for the current client session
CodePage	Gets and sets the code page used when reading and writing text
ExternalAddress	Return the external IP address assigned to the local system
IsActive	Determine if the server has been started
IsBlocked	Determine if the control is blocked performing an operation
IsClosed	Determine if the current client connection has been closed by the remote host
IsInitialized	Determine if the control has been initialized
IsListening	Determine if the server is listening for connections
IsReadable	Return if data can be read from the current client socket without blocking
IsWritable	Return if data can be written to the current client socket without blocking
KeepAlive	Set or return if keep-alive packets are sent to connected clients
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error that occurred
MaxClients	Gets and sets the maximum number of clients that can connect to the server
MemoryUsage	Gets the amount of memory allocated for the server and all client sessions
NoDelay	Enable or disable the Nagle algorithm
Priority	Gets and sets the priority assigned to the server
ReuseAddress	Set or return if the server address can be reused

Secure	Set or return if client connections are encrypted using the TLS security protocol.
SecureProtocol	Gets and sets the security protocol used to establish a secure connection
ServerAddress	Gets and sets the address that will be used by the server to listen for connections
ServerHandle	Return the handle to the socket created to listen for client connections
ServerName	Return the fully qualified domain name of the local system
ServerPort	Gets and sets the port number that will be used by the server to listen for connections
ServerThread	Return the thread ID for the server
StackSize	Gets and sets the size of the stack allocated for threads created by the server
ThrowError	Enables or disables error handling by the application
Timeout	Gets and sets the amount of time until a blocking operation fails
Trace	Enable or disable socket function level tracing
TraceFile	Specify the socket function trace output file
TraceFlags	Gets and sets the socket function tracing flags
Version	Return the current version of the control

AdapterAddress Property

Returns the IP address associated with the specified network adapter.

Syntax

object.AdapterAddress(Index)

Remarks

The **AdapterAddress** property array returns the IP addresses that are associated with the local network or remote dial-up network adapters configured on the system. The **AdapterCount** property can be used to determine the number of adapters that are available.

Multihomed systems with more than one local network adapter, or a combination of local and dial-up adapters will not be listed in a specific order. An application should not make the assumption that the address returned by **AdapterAddress(0)** always refers to a local network adapter.

Note that it is possible that the **AdapterCount** property will return 0, and **AdapterAddress(0)** will return an empty string. This indicates that the system does not have a physical network adapter with an assigned IP address, and there are no dial-up networking connections currently active. If a dial-up networking connection is established at some later point, the **AdapterCount** property will change to 1, and the **AdapterAddress(0)** property will return the IP address allocated for that connection.

When using Visual Studio .NET, you must use the accessor method **get_AdapterAddress** instead of the property name, otherwise an error will be returned indicating that it not a member of the control class.

Data Type

String

See Also

[AdapterCount Property](#)

AdapterCount Property

Returns the number of available local and remote network adapters.

Syntax

object.AdapterCount

Remarks

The **AdapterCount** property returns the number of local and remote dial-up networking adapters available on the local system. This value can be used in conjunction with the **AdapterAddress** property array to enumerate the IP addresses assigned to the various network adapters.

Note that it is possible that the **AdapterCount** property will return 0, and **AdapterAddress**(0) will return an empty string. This indicates that the system does not have a physical network adapter with an assigned IP address, and there are no dial-up networking connections currently active. If a dial-up networking connection is established at some later point, the **AdapterCount** property will change to 1, and the **AdapterAddress**(0) property will return IP address allocated for that connection.

Data Type

Integer (Int32)

See Also

[AdapterAddress Property](#)

Backlog Property

Gets and sets the number of client connections that may be queued by the server.

Syntax

object.Backlog [= *backlog*]

Remarks

The **Backlog** property specifies the maximum size of the queue used to manage pending connections to the service. If the property is set to value which exceeds the maximum size for the underlying service provider, it will be silently adjusted to the nearest legal value. There is no standard way to determine what the maximum backlog value is.

This property should be set to the desired value before the **Start** method is called. The default backlog value is 5 on all Windows platforms. The Windows Server platforms support a maximum backlog value of 200.

Note that this property does not specify the total number of connections that the server application may accept. It only specifies the size of the backlog queue which is used to manage pending client connections. Once the client connection has been accepted, it is removed from the queue. Set the **MaxClients** property to specify the maximum number of clients that may connect with the server.

Data Type

Integer (Int32)

See Also

[IsListening Property](#), [MaxClients Property](#), [Start Method](#), [Throttle Method](#), [OnAccept Event](#)

ByteOrder Property

Gets and sets the byte order in which integer data will be written to and read from the socket.

Syntax

object.ByteOrder [= 0 | 1]

Remarks

The **ByteOrder** property is used to specify how 16-bit (short) integer and 32-bit (long) integer data is written to and read from the socket. The default value for this property is 0, which specifies that integers should be written in the native byte order for the local machine. A value of 1 indicates that integers should be written in network byte order.

When applications write integer values on a socket (instead of string representations of those values), they should typically be converted to network byte order before they are sent. Likewise, when an integer value is read, it should then be converted from the network byte order back to the byte order used by the local machine. The native byte order, also called the host byte order, should only be used if it can be assured that both the sender and the receiver are running on an identical or compatible machine architectures (for example, if both systems are Intel-based).

This property will affect how data is read by the **Read** method and by the **Write** method, if the Variant data that is being read or written is recognized as integer data.

Data Type

Integer (Int32)

CertificateName Property

Gets and sets the common name for the server certificate.

Syntax

object.CertificateName [= *name*]

Remarks

This property sets the common name or friendly name of the certificate that should be used when starting a secure server. If the **Secure** property is set to True, this property must be specify a valid certificate name. The certificate must have a private key associated with it, otherwise client connections will fail because the control will be unable to create a security context for the session.

Certificates may be installed and viewed on the local system using the Certificate Manager that is included with the Windows operating system. For more information, refer to the documentation for the Microsoft Management Console.

Data Type

String

See Also

[CertificateStore Property](#), [Secure Property](#)

CertificatePassword Property

Gets and sets the password associated with the server certificate.

Syntax

object.CertificatePassword [= *password*]

Remarks

This property sets the password that should be used to access a certificate in the specified certificate store. It is only required when the **CertificateStore** property specifies a file that contains a certificate and private key in PKCS #12 format.

Data Type

String

See Also

[CertificateName Property](#), [CertificateStore Property](#), [Secure Property](#)

CertificateStore Property

Gets and sets the name of the server certificate store or file.

Syntax

object.CertificateStore [= *store*]

Remarks

This property sets the name of the certificate store that contains the server certificate that should be used when starting a server with security enabled. The certificate may either be stored in the registry or in a file. If the certificate is stored in the registry, then this property should be set to one of the following predefined values:

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. If a certificate store is not specified, this is the default value that is used.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

In most cases the client certificate will be installed in the user's personal certificate store, and therefore it is not necessary to set this property value because that is the default location that will be used to search for the certificate. This property is only used if the **CertificateName** property is also set to a valid certificate name.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU" for the current user, or "HKLM" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, it will default to the certificate store for the current user.

This property may also be used to specify a file that contains the client certificate. In this case, the property should specify the full path to the file and must contain both the certificate and private key in PKCS #12 format. If the file is protected by a password, the **CertificatePassword** property must also be set to specify the password.

Data Type

String

See Also

[CertificateName Property](#), [CertificatePassword Property](#), [Secure Property](#)

CertificateUser Property

Gets and sets the user that owns the server certificate.

Syntax

object.CertificateUser [= *username*]

Remarks

This property sets the name of the user that owns the server certificate that will be used. If this property is not set, the certificate store for the current user will be used when searching for the certificate. If this property is used to specify another user, the server process must have the appropriate permission to access the registry location that contains the client certificate. On Windows Vista and later versions of the operating system, this requires that the process run with elevated privileges.

Data Type

String

See Also

[CertificateName Property](#), [CertificateStore Property](#), [Secure Property](#)

ClientAddress Property

Return the Internet address of the current client connection.

Syntax

object.ClientAddress

Remarks

The **ClientAddress** property returns the address of the current client session which has connected to the server. This property only returns a meaningful value inside an event handler such as **OnAccept** or **OnConnect**.

If this property is accessed inside an **OnAccept** event handler, it will return the address of the client that is requesting the connection. The server application may use this information to determine if it wishes to accept or reject the client connection.

Data Type

String

See Also

[ClientHost Property](#), [ClientPort Property](#), [ServerAddress Property](#), [OnAccept Event](#), [OnConnect Event](#)

ClientCount Property

Return the number of active client sessions connected to the server.

Syntax

object.ClientCount

Remarks

The **ClientCount** read-only property returns the number of active client sessions that have been established with the server.

The value returned by this property does not include clients that are in the process of terminating. For example, if the **Suspend** method is called to suspend the server and terminate all of the client connections, each client is signaled to disconnect from the server and the active client count is immediately set to zero. Once a client has been signaled to disconnect, it is no longer considered to be an active client connection even if that session does not terminate immediately. This means that you cannot use this property value to determine the number of clients in the process of disconnecting from the server or when all clients have disconnected.

To determine when all clients have disconnected from the server after the **Suspend** or **Restart** method has been called, you must implement an **OnIdle** event handler. This event occurs after the last active client session has terminated.

Data Type

Integer (Int32)

See Also

[ClientHandle Property](#), [Restart Method](#), [Suspend Method](#), [OnDisconnect Event](#), [OnIdle Event](#)

ClientHandle Property

Return the socket handle associated with a specific client session.

Syntax

object.ClientHandle(*Index*)

Remarks

The **ClientHandle** property is read-only, zero-based property array that returns the socket handle allocated for the client session specified by the *Index* parameter. An exception will be thrown if the index value exceeds the maximum number of active client sessions. To determine the number of clients that are currently connected to the server, use the **ClientCount** property.

You should always check the value of the **ClientCount** property prior to enumerating through the client connections using the **ClientHandle** property array. Never assume that a particular client session will always be found in the same position in the property array. The socket handles returned by the property array can be used in conjunction with the **Read** and **Write** methods to exchange data with a particular client session outside of an event handler.

Data Type

Integer (Int32)

See Also

[ClientHandle Property](#), [Disconnect Method](#), [FindClient Method](#), [Read Method](#), [Write Method](#), [OnConnect Event](#), [OnRead Event](#)

ClientHost Property

Return the hostname for the current client session.

Syntax

object.ClientHost

Remarks

The **ClientHost** property returns the hostname of the current client session which has established a connection with the server. This property value is only meaningful when accessed within an event handler, such as the **OnConnect** event.

Accessing this property causes the control to perform a blocking reverse DNS lookup, attempting to match the client Internet address with a hostname. Not all addresses have a reverse DNS record, in which case this property will return an empty string. It is recommended that most applications use the value of the **ClientAddress** property rather than use the **ClientHost** property to distinguish between client connections.

Data Type

String

See Also

[ClientAddress Property](#), [ClientPort Property](#), [ServerAddress Property](#), [OnAccept Event](#), [OnConnect Event](#)

ClientId Property

Return a unique identifier for the current client session.

Syntax

object.ClientId

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value. This value can be used by the application to identify that client session, and is different than the socket handle allocated for the client. While it is possible for a client socket handle to be reused by the operating system, client IDs are unique throughout the life of the server session and are never duplicated.

It is important to note that the actual value of the client ID should be considered opaque. It is only guaranteed that the value will be greater than zero, and that it will be unique to the client session.

This property only returns a meaningful value when accessed from within an event handler, or a function that has been called from within an event handler.

Data Type

Integer (Int32)

See Also

[ClientAddress Property](#), [ClientHost Property](#), [ClientName Property](#), [ServerAddress Property](#), [ServerPort Property](#)

ClientName Property

Gets and sets a unique string moniker that is associated with the current client session.

Syntax

object.ClientName [= *moniker*]

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session by setting the **ClientName** property from within a class event handler such as the **OnConnect** event.

Monikers are not case-sensitive, and they must be unique so that no client socket for a particular server can have the same moniker. The maximum length for a moniker is 127 characters.

This property only returns a meaningful value when accessed from within an event handler, or a function that has been called from within an event handler.

Data Type

String

See Also

[ClientAddress Property](#), [ClientId Property](#), [ServerAddress Property](#), [ServerPort Property](#)

ClientPort Property

Return the port number of the current client connection.

Syntax

object.**ClientPort**

Remarks

The **ClientPort** property returns the port number that the current client has used when establishing a connection with the server. This property value is only meaningful when accessed within an event handler such as the **OnConnect** event.

Data Type

Integer (Int32)

See Also

[ClientAddress Property](#), [ClientHost Property](#), [ServerAddress Property](#), [ServerPort Property](#)

ClientThread Property

Return the thread ID for the active client session.

Syntax

object.ClientThread

Remarks

The **ClientThread** property returns the thread ID for the current client session. Until the thread terminates, the thread identifier uniquely identifies the thread throughout the system. This property only returns a meaningful value when accessed from within an event handler, or a function that has been called from within an event handler.

The thread ID can be used with Windows API functions such as **OpenThread**. Exercise caution when using thread-related functions, interfering with the normal operation of the thread can have unexpected results. You should never use this property value to obtain a thread handle and then call the **TerminateThread** function to terminate a client session. This will prevent the thread from releasing the resources that were allocated for the session and can leave the server in an unstable state. To terminate a client session, use the **Disconnect** method.

Data Type

Integer (Int32)

See Also

[ClientId Property](#), [ServerThread Property](#)

CodePage Property

Gets and sets the code page used when reading and writing text.

Syntax

object.CodePage [= *value*]

Remarks

The **CodePage** property is an integer value which specifies how strings are encoded when data is sent or received. Any valid code page identifier may be specified. Some common values are:

Value	Description
0	Text sent and received using a string should be converted using the ANSI code page for the current locale.
1	Text sent and received using a string should be converted using the system default OEM code page. The OEM code page typically contains characters that are used by console applications and are based on character sets commonly used by MS-DOS. You should not use this code page unless you know the server is sending text which includes OEM characters.
1252	Text sent and received using a string should be converted using the Windows ANSI code page for western European languages. This code page is commonly used by legacy Windows applications for English and some other western languages. It should be noted that while this code page is similar to ISO 8859-1 character encoding, it is not identical.
28591	Text sent and received using a string should be converted using the ISO 8859-1 code page for western European languages. This code page is commonly referred to as Latin-1 and is similar to the Windows 1252 code page.
65000	Data that is sent and received using a string should be converted using UTF-7 encoding. If this code page is specified, data written to the socket will be encoded as UTF-7 encoded Unicode. All data received from the server will be converted from UTF-7. It is not recommended that you use this code page unless you know that the remote host is sending UTF-7 encoded text.
65001	Data that is sent and received using a string should be converted using UTF-8 encoding. If this code page is specified, data written to the socket will be encoded as UTF-8 encoded Unicode. All data received from the server will be converted from UTF-8 to UTF-16 Unicode. Because UTF-8 is backwards compatible with the ASCII character set, it is safe to use this encoding option when sending and receiving ASCII text.

A complete list of available [code page identifiers](#) can be found in Microsoft's documentation for the Win32 API.

All data which is exchanged over a socket is sent and received as 8-bit bytes, typically referred to as "octets" in networking terminology. However, the internal string type used by ActiveX controls are Unicode where each character is represented by 16 bits. To send and receive data using strings, these Unicode strings are converted to a stream of bytes.

By default, strings are converted to an array of bytes using the code page for the current locale, mapping the 16-bit Unicode characters to bytes. Similarly, when reading data from the socket into a

string buffer, the stream of bytes received from the remote host are converted to Unicode before they are returned to your application.

If you are exchanging text with another system and it appears to be corrupted or characters are being replaced with question marks or other symbols, it is likely the system is sending text which is using a different character encoding. Most services use UTF-8 encoding to represent non-ASCII characters and selecting the UTF-8 code page will typically resolve the issue.



Strings are only guaranteed to be safe when sending and receiving text. Using a string data type is not recommended when reading or writing binary data to a socket. If possible, you should always use a byte array as the buffer parameter for the **Read** and **Write** methods whenever you are exchanging binary data.

For backwards compatibility, the control defaults to using the code page for the current locale. This property value directly corresponds to Windows code page identifiers, and will accept any valid code page in addition to the values listed above. Setting this property to an invalid code page will result in an error.

Data Type

Integer (Int32)

See Also

[Read Method](#), [ReadLine Method](#), [Write Method](#), [WriteLine Method](#)

ExternalAddress Property

Return the external IP address for the local system.

Syntax

object.ExternalAddress

Remarks

The **ExternalAddress** property returns the IP address assigned to the router that connects the local host to the Internet. This is typically used by an application executing on a system in a local network that uses a router which performs Network Address Translation (NAT). In that network configuration, the **LocalAddress** property will only return the IP address for the local system on the LAN side of the network unless a connection has already been established to a remote host. The **ExternalAddress** property can be used to determine the IP address assigned to the router on the Internet side of the connection and can be particularly useful for servers running on a system behind a NAT router.

Using this property requires that you have an active connection to the Internet; checking the value of this property on a system that uses dial-up networking may cause the operating system to automatically connect to the Internet service provider. The control may be unable to determine the external IP address for the local host for a number of reasons, particularly if the system is behind a firewall or uses a proxy server that restricts access to external sites on the Internet. If the external address for the local host cannot be determined, the property will return an empty string.

If the control is able to obtain a valid external address for the local host, that address will be cached for sixty minutes. Because dial-up connections typically have different IP addresses assigned to them each time the system is connected to the Internet, it is recommended that this property only be used in conjunction with broadband connections using a NAT router. Checking this property value may cause the thread to block until the external IP address can be resolved.

Data Type

String

See Also

[ClientAddress Property](#), [ServerAddress Property](#)

IsActive Property

Determine if the server has been started.

Syntax

object.IsActive

Remarks

The **IsActive** property returns **True** if the server has been started using the **Start** method. If the server has not been started, the property will return **False**.

To determine if the server is accepting client connections, use the **IsListening** property. This property will only indicate if the server has been started. For example, if the server has been suspended using the **Suspend** method, this property will return a value of **True**, while the **IsListening** property will return a value of **False**.

An application should not depend on this property returning **False** immediately after the **Stop** method has been called to shutdown the server. This property will continue to return **True** until all clients have disconnected from the server and the server thread has terminated. To determine when the server has stopped, implement a handler for the **OnStop** event.

Data Type

Boolean

See Also

[IsListening Property](#), [Start Method](#), [Stop Method](#), [OnStop Event](#)

IsBlocked Property

Determine if the control is blocked performing an operation.

Syntax

object.IsBlocked

Remarks

The **IsBlocked** property returns True if the control is blocked performing an operation. If the **IsBlocked** property returns False, this means there are no blocking operations on the current thread at that time. If the property returns True, this tells you that you can't proceed with a socket operation. However, if the property returns False this does not guarantee that the next socket operation will not fail with a **swErrorOperationWouldBlock** or **swErrorOperationInProgress** error. The application should treat these errors as recoverable, and should be prepared to retry operations that result in them.

Note that this property will return True if there is *any* blocking operation being performed by the application, regardless of whether the control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

[LastError Property](#)

IsClosed Property

Determine if the current client connection has been closed by the remote host.

Syntax

object.IsClosed

Remarks

The **IsClosed** property returns True if the current client connection has been closed by the remote host. The value of this property is only meaningful inside an event handler such as **OnRead**.

Data Type

Boolean

See Also

[IsReadable Property](#), [IsWritable Property](#), [OnConnect Event](#), [OnRead Event](#), [OnWrite Event](#)

IsInitialized Property

Determine if the control has been initialized.

Syntax

object.IsInitialized

Remarks

The **IsInitialized** property is used to determine if the current instance of the control has been initialized properly. Normally this is done automatically when the control is loaded, however there are circumstances where the control may not be able to initialize itself. If this property returns **False**, the application must call the **Initialize** method to initialize the control before performing any other operation.

The most common reason that the control may not initialize correctly is that no valid development or runtime license key can be found or the license key that was provided is invalid. It may also indicate a problem with the system configuration or user access rights, such as not being able to load the required networking libraries or not being able to access the system registry.

Data Type

Boolean

See Also

[Initialize Method](#)

IsListening Property

Determine if the server is listening for client connections.

Syntax

object.IsListening

Remarks

The **IsListening** property returns True if the server is listening for connections after the **Start** method has been called. If the server has not been started, is not yet accepting client connections or has been suspended, this property will return False.

When a server is started, the control starts a background thread which creates the listening socket and begins waiting for incoming client connections. This property will only return True once the server thread has started executing, so it may not return a value of True immediately after the **Start** method has been called. To determine the status of the server at any time, check the value of the **State** property.

Data Type

Boolean

See Also

[IsActive Property](#), [Start Method](#), [Stop Method](#)

IsReadable Property

Return if data can be read from the current client socket without blocking.

Syntax

object.IsReadable

Remarks

The **IsReadable** property returns True if data can be read from the current client socket without blocking. This property can be checked before the application attempts to read the socket, preventing an error. The value of this property is only meaningful inside an event handler such as **OnRead**.

Data Type

Boolean

See Also

[IsClosed Property](#), [IsWritable Property](#), [Peek Method](#), [Read Method](#), [OnRead Event](#)

IsWritable Property

Return if data can be written to the current client socket without blocking.

Syntax

object.IsWritable

Remarks

The **IsWritable** property returns True if data can be written to the current client socket without blocking. If the **IsWritable** property returns False, this means that the application cannot write to the socket at that time. However, if the property returns True, this does not guarantee that you will be able to write to the socket without an error. The next socket operation may result in a **swErrorOperationWouldBlock** or **swErrorOperationInProgress** error. The application should treat these errors as recoverable, and should be prepared to retry operations that result in them.

The value of this property is only meaningful inside an event handler such as **OnRead** or **OnWrite**.

Data Type

Boolean

See Also

[IsClosed Property](#), [IsReadable Property](#), [OnWrite Event](#)

KeepAlive Property

Set or return if keep-alive packets are sent to connected clients.

Syntax

object.KeepAlive [= { True | False }]

Remarks

Setting the **KeepAlive** property to a value of True indicates that packets are to be sent to connected clients when no data is being exchanged to keep the connection active.

The default interval at which these packets are sent is typically two hours and cannot be modified using the control. Consult the Windows system administration documentation for information on how to change the default keep-alive interval.

Data Type

Boolean

See Also

[NoDelay Property](#), [ReuseAddress Property](#)

LastError Property

Gets and sets the last error that occurred on the control.

Syntax

object.**LastError** [= *errorcode*]

Remarks

The **LastError** property can be read to determine the last error that occurred for this control. If a value is assigned to this property, it must either be zero (to clear the error) or a valid error code for the control.

Data Type

Integer (Int32)

See Also

[LastErrorString Property](#), [ThrowError Property](#), [OnError Event](#)

LastErrorString Property

Return a description of the last error that occurred.

Syntax

object.LastErrorString

Remarks

The **LastErrorString** property returns a string that contains a description of the last error that occurred.

Data Type

String

See Also

[LastError Property](#), [ThrowError Property](#), [OnError Event](#)

MaxClients Property

Gets and sets the maximum number of clients that can connect to the server.

Syntax

object.MaxClients [= *clients*]

Remarks

The **MaxClients** property specifies the maximum number of client connections that will be accepted by the server. Once the maximum number of connections has been established, the server will reject any subsequent connections until the number of active client connections drops below the specified value. A value of zero specifies that there should be no limit on the number of clients.

Changing the value of this property while a server is actively listening for connections will modify the maximum number of client connections permitted, but it will not affect connections that have already been established.

By default, there are no limits on the number of client connections or the connection rate when a server is started. Use the **Throttle** method to change the maximum number of client connections per IP address or the overall connection rate threshold for the server.

It is important to note that regardless of the maximum number of clients specified by this property, the actual number of client connections that can be managed by the server depends on the number of sockets that can be allocated from the operating system. The amount of physical memory installed on the system affects the number of connections that can be maintained because each connection allocates memory for the socket context from the non-paged memory pool.

Data Type

Integer (Int32)

See Also

[Backlog Property](#), [Timeout Property](#), [Start Method](#), [Throttle Method](#)

MemoryUsage Property

Gets the amount of memory allocated for the server and all client sessions.

Syntax

object.MemoryUsage

Remarks

This read-only property returns the amount of memory allocated by the server and all active client sessions. It enumerates all memory allocations made by the server process and client session threads, returning the total number of bytes allocated for the server process. This value reflects the amount of memory explicitly allocated by this control and does not reflect the total working set size of the process, or memory allocated by any other components or libraries.

Getting the value of this property forces the server into a locked state, and all client sessions will block while the memory usage is being calculated. Because this enumerates all heaps allocated for the server process, it can be an expensive operation, particularly when there are a large number of active clients connected to the server. Frequently checking the value of this property can significantly degrade the performance of the server. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active client sessions. If the value returned by this property remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Data Type

Double

See Also

[StackSize Property](#), [Start Method](#)

NoDelay Property

Enable or disable the Nagle algorithm.

Syntax

object.NoDelay [= { True | **False** }]

Remarks

The **NoDelay** property is used to enable or disable the Nagle algorithm, which buffers unacknowledged data and ensures that a full-size packet can be sent to the remote host. By default this property value is set to False, which enables the Nagle algorithm (in other words, the data being written may not actually be sent until it is optimal to do so). Setting this property to True disables the Nagle algorithm, minimizing the time delays between the data packets being sent.

This property should be set to True only if it is absolutely required and the implications of doing so are understood. Disabling the Nagle algorithm can have a significant negative impact on the performance of the server.

Data Type

Boolean

See Also

[KeepAlive Property](#), [ReuseAddress Property](#)

Priority Property

Gets and sets the priority assigned to the server.

Syntax

object.Priority [= *priority*]

Remarks

The **Priority** property can be used to control the processor usage, memory and network bandwidth allocated by the server for client sessions. One of the following values may be specified:

Value	Description
swPriorityBackground	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. Each client thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
swPriorityLow	This priority lowers the overall resource utilization for the client session and meters the processor utilization for the client session. Each client thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
swPriorityNormal	The default priority which balances resource and processor utilization. It is recommended that most applications use this priority.
swPriorityHigh	This priority increases the overall resource utilization for each client session and their threads will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
swPriorityCritical	This priority can significantly increase processor, memory and network utilization. Each client thread will be given higher scheduling priority and will be more responsive to network events. It is not recommended that this priority be used on a system with a single processor.

The **swPriorityNormal** priority balances resource and network bandwidth utilization while ensuring that a single-threaded server application remains responsive to the user. Lower priorities reduce the overall resource utilization of the server at the expense of throughput.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

Data Type

Integer (Int32)

See Also

ReuseAddress Property

Set or return if the local address can be reused by the server.

Syntax

`object.ReuseAddress [= { True | False } ]`

Remarks

The **ReuseAddress** property is used to determine if the local address and port number can be reused when starting a new instance of the server. Setting this property to True enables a server application to listen for connections using the specified address and port number even if they were in use recently. This is typically used to enable the server to close the listening socket and immediately reopen it without getting an error that the address is in use.

When a listening socket closed, the socket will normally go into a TIME-WAIT state where the local address and port number cannot be immediately reused. A consequence of this is that calling the **Stop** method immediately followed by the **Start** method using the same address and port number values may result in an error indicating that the specified address is already in use. By setting this property to True, that error is avoided and the listening socket can be created immediately without waiting for the TIME-WAIT period to elapse. Note that calling the **Restart** method allows the local address and port number to be reused, regardless of this property value.

If you wish to determine if a local port number is already in use by another application, set this property to false and attempt to start a server using that port number. If another application is already using that port number, an error will be generated indicating that the address is in use and the server could not be started.

Data Type

Boolean

See Also

[ServerAddress Property](#), [ServerPort Property](#), [Restart Method](#), [Start Method](#), [Stop Method](#)

Secure Property

Set or return if client connections are encrypted using the TLS protocol.

Syntax

object.**Secure** [= { True | False }]

Remarks

The **Secure** property determines if client connections are encrypted using the Transport Layer Security (TLS) protocol. The default value for this property is False, which specifies that clients will use a standard, unencrypted connection to the server. To enable secure connections, the application should set this property value to True prior to calling the **Start** method.

When secure connections are enabled, the server will accept the client connection and then wait for the client to initiate the handshake where both the client and server negotiate the various encryption options available. This process is handled automatically by the server, and all that is required is that the application specify the server certificate which should be used. This is done by setting the **CertificateName** property, and optionally the **CertificateStore** property if required.

It is recommended that the application use exception handling to catch any errors that may occur when changing the value of this property. If the control is unable to initialize the Windows security libraries, an exception will be thrown when this property value is modified.

Data Type

Boolean

See Also

[CertificateName Property](#), [CertificateStore Property](#), [SecureProtocol Property](#), [Start Method](#)

SecureProtocol Property

Gets and sets the security protocol used to accept a secure connection.

Syntax

object.SecureProtocol [= *protocol*]

Remarks

The **SecureProtocol** property can be used to specify the security protocol to be used when accepting a secure connection. By default, the control will attempt to use TLS 1.3 to establish the connection. If TLS 1.3 is not supported, TLS 1.2 will be used. The appropriate protocol is automatically selected based on the capabilities of both the client and server.

It is recommended that you only change this property value if you fully understand the implications of doing so. Assigning a value to this property will override the default and force the control to attempt to use only the protocol specified. One or more of the following values may be used:

Value	Description
stProtocolNone	No security protocol has been selected. A secure connection has not been established.
stProtocolTLS10	The TLS 1.0 protocol should be used when accepting a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
stProtocolTLS11	The TLS 1.1 protocol should be used when accepting a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
stProtocolTLS12	The TLS 1.2 protocol should be used when accepting a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
stProtocolTLS13	The TLS 1.3 protocol should be used when accepting a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

Multiple security protocols may be specified by combining them using a bitwise Or operator. Attempting to set this property after the server has been started will result in an exception being thrown. This property should only be set after setting the **Secure** property to True and before calling the **Start** method.

Data Type

Integer (Int32)

See Also

[CertificateName Property](#), [CertificateStore Property](#), [Secure Property](#), [Start Method](#)

ServerAddress Property

Gets and sets the address that will be used by the server to listen for connections.

Syntax

object.ServerAddress [= *address*]

Remarks

The **ServerAddress** property is used to specify the default address that the server will use when listening for connections. Setting this property to the value 0.0.0.0 or an empty string indicates that the server should listen for client connections using any valid network interface. If an address is specified, it must be a valid Internet address that is bound to a network adapter configured on the local system. Clients will only be able to connect to the server using that specific address.

It is common to set this property to the value 127.0.0.1 for testing purposes. It is a non-routable address that specifies the local system, and most software firewalls are configured so they do not block applications using this address.

Data Type

String

See Also

[ExternalAddress Property](#), [ServerName Property](#), [ServerPort Property](#), [Start Method](#)

ServerName Property

Return the fully qualified domain name of the local system.

Syntax

object.ServerName

Remarks

The **ServerName** read-only property returns the fully qualified domain name of the local system. This consists of the local computer name and its domain name. The actual value returned depends on the system configuration. If no domain has been specified for the system, then only the machine name will be returned.

Data Type

String

See Also

[ServerAddress Property](#), [ServerPort Property](#), [Resolve Method](#)

ServerPort Property

Gets and sets the port number that will be used by the server to listen for connections.

Syntax

object.ServerPort [= *port*]

Remarks

The **ServerPort** property is used to set the port number that server will use to listen for incoming client connections. Valid port numbers are in the range of 1 to 65535. It is recommended that most custom servers specify a port number larger than 5000 to avoid potential conflicts with standard Internet services and ephemeral ports used by client applications.

If a port number is specified that is already in use by another application, the **OnError** event will fire and the background server thread will terminate. To enable a server to be stopped and immediately restarted using the same address and port number, make sure that the **ReuseAddress** property is set to a value of True.

Data Type

Integer (Int32)

See Also

[ReuseAddress Property](#), [ServerAddress Property](#), [ServerName Property](#), [Start Method](#)

ServerThread Property

Return the thread ID for the server.

Syntax

object.ServerThread

Remarks

The **ServerThread** property returns the thread ID for the active server. Until the thread terminates, the thread identifier uniquely identifies the thread throughout the system. If there is no active server, this property will return a value of zero.

Data Type

Integer (Int32)

See Also

[ClientAddress Property](#), [ClientThread Property](#), [ServerAddress Property](#), [ServerPort Property](#)

StackSize Property

Gets and sets the size of the stack allocated for threads created by the server.

Syntax

object.**StackSize** [= *bytes*]

Remarks

The **StackSize** property returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change this value unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Data Type

Integer (Int32)

See Also

[MemoryUsage Property](#), [Start Method](#)

ThrowError Property

Enables or disables error handling by the application.

Syntax

object.ThrowError = { True | False }

Remarks

Error handling for methods can be performed using either return codes or exceptions, depending on the value of this property.

If the **ThrowError** property is set to False, methods will return a status value indicating success or failure. Applications should check the return value of each method and use the **LastError** and **LastErrorString** properties to determine the cause of any error.

If the **ThrowError** property is set to True, errors occurring within the control will be raised to the host application as ActiveX exceptions. In addition, the **OnError** event will fire. This allows applications to use language-specific exception handling mechanisms, such as Visual Basic error handling, instead of checking individual return values after each method call.

Even when the **ThrowError** property is set to True, the **LastError** and **LastErrorString** properties will still be updated with information about the most recent error.

If an error occurs while accessing a property value, an exception will be raised regardless of the value of the **ThrowError** property. However, the **OnError** event will not be fired for property access errors.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
Dim nError As Long

' The control will not raise an exception when an error occurs
Server1.ThrowError = False

' Start the server
nError = Server1.Start()

' If the method returns an error code, then display a message box
' and exit the subroutine
If nError > 0 Then
    MsgBox Server1.LastErrorString, vbExclamation
    Exit Sub
End If
```

The following example handles errors by generating an exception:

```
On Error GoTo Failed

' The control will raise an exception when an error occurs
Server1.ThrowError = True

' Start the server
Server1.Start
Exit Sub

Failed:
```

```
' If the method fails, code execution will resume at this label  
Failed:  
MsgBox Err.Description, vbExclamation  
Exit Sub
```

See Also

[LastError Property](#), [OnError Event](#)

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.Timeout [= *seconds*]

Remarks

Setting this property specifies the number of seconds until a blocking network operation fails and the control returns an error.

Data Type

Integer (Int32)

See Also

[LastError Property](#), [OnError Event](#), [OnTimeout Event](#)

Trace Property

Enable or disable network function level logging.

Syntax

`object.Trace` [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of network function calls. When enabled, each function call is logged to a file, including the function parameters, return value and error code if applicable. This facility can be enabled and disabled at run time, and the trace log file can be specified by setting the **TraceFile** property. All function calls that are being logged are appended to the trace file, if it exists. If no log file exists when tracing is enabled, the file is created.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

Data Type

Boolean

See Also

[TraceFile Property](#), [TraceFlags Property](#)

TraceFile Property

Gets and sets the name of the file to use when logging network function calls.

Syntax

`object.TraceFile [= filename ]`

Remarks

The **TraceFile** property is used to specify the name of the file that is created when logging is enabled. If this property is set to an empty string (the default value), then a file named **SocketTools.log** is created in the current user's Documents folder. It is recommended that you always specify a complete path to the log file. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

If the file exists, the output is appended to the file, otherwise the file is created. Since logging is enabled per-process, the log file is shared by all instances of the controls being used. If you create multiple instances of the control, you only need to enable logging for a single instance. Because log files can grow very quickly, even with modest applications, it is recommended that you delete the file when it is no longer needed.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "0x". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxx:xxxx:xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, the control is attempting to connect to a system with the IPv4 address 192.0.0.1 on port 1234.

If the log file cannot be created, or the user does not have permission to modify an existing file, the error is silently ignored and no output will be generated.

Data Type

String

See Also

TraceFlags Property

Gets and sets the network function logging flags.

Syntax

object.TraceFlags [= *flags*]

Remarks

The **TraceFlags** property is used to specify the type of information written to the trace file when socket function tracing is enabled. The following values may be used:

Value	Description
dnsTraceInfo	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
dnsTraceError	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
dnsTraceWarning	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
dnsTraceHexDump	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
dnsTraceTruncate	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.

Since logging is enabled per-process, the trace flags are shared by all instances of the control being used. If multiple controls have tracing enabled, the **TraceFlags** property should be set to the same value for each control. Changing the trace flags for any one instance of the control will affect the logging performed for all other instances used by the application.

Warnings are generated when a non-fatal error is returned by a Windows Sockets function. For example, if data is being written through the control and an error indicating that the operation would block is returned, only a warning is logged since the application simply needs to attempt to write the data at a later time.

Data Type

Integer (Int32)

See Also

[Trace Property](#), [TraceFile Property](#)

Version Property

Return the current version of the control.

Syntax

object.**Version**

Remarks

The **Version** property returns the current version of the object. This can be used by an application for validation purposes.

Data Type

String

Internet Server Control Methods

Method	Description
Abort	Abort the specified client session, terminating its connection to the server
Broadcast	Broadcast data to all active clients connected to the server
Cancel	Cancels the current blocking network operation
Disconnect	Disconnect the specified client session from the server
FindClient	Return the socket handle for the client session with the specified moniker or client ID
Initialize	Initialize the control and validate the runtime license key
Peek	Read data from the specified client session, but do not remove it from the socket buffer
Read	Read data from the specified client session
ReadLine	Read a line of data from the specified client session, storing it in a string buffer
Reject	Reject a pending client connection
Reset	Reset the internal state of the control, stopping the server and terminating all client connections
Resolve	Resolves a host name to a host IP address
Restart	Restart the server, terminating all active client connections
Resume	Resume accepting new client connections
Start	Start listening for client connections on the specified IP address and port number
Stop	Stop listening for new client connections and terminate all client sessions
Suspend	Suspend accepting new client connections
Throttle	Limit the maximum number of client connections, connections per IP address and connection rate
Uninitialize	Uninitialize the control and release any system resources that were allocated
Write	Write data to the specified client session
WriteLine	Write a line of data to the specified client session, terminated with a carriage-return and linefeed

Abort Method

Abort the specified client session, terminating its connection to the server.

Syntax

object.Abort(Handle)

Parameters

Handle

An integer value that specifies the handle to the client session.

Return Value

A value of zero is returned if the connection was aborted successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Abort** method immediately closes the specified client socket, terminating its connection to the server. Any queued data in the socket's send and receive buffers will be discarded, and the client may terminate abnormally unless it is designed to handle aborted connections. It is not recommended that you use this method unless you understand the implications of doing so. To gracefully terminate the client connection, use the **Disconnect** method.

See Also

[Disconnect Method](#), [Stop Method](#)

Broadcast Method

Broadcast data to all active clients connected to the server.

Syntax

object.Broadcast(*Buffer*, [*Length*])

Parameters

Buffer

A buffer variable that contains the data to be written to the server. If the variable is a **String** type, then the data will be written as a string of characters. This is the most appropriate data type to use if the server expects text data that consists of printable characters. If the server is expecting binary data, it is recommended that a **Byte** array be used instead.

Length

A numeric value which specifies the number of bytes to write. Its maximum value is $2^{31}-1 = 2147483647$. If a value is specified for this argument and it is greater than the actual size of the buffer, then the **Length** argument will be ignored and the entire contents of the buffer will be written. If the argument is omitted, then the maximum number of bytes to write is determined by the size of the buffer.

Return Value

This method returns the number of clients that the data was broadcast to. A return value of -1 indicates an error condition, and the value of the **LastError** property will indicate the cause of the failure.

Remarks

The **Broadcast** method sends the data in **Buffer** to all clients connected to the server. If this method is called inside a server event handler, the message is broadcast to all clients except for the current, active client that is processing the event notification. If this method is called outside of an event handler, the data is broadcast to all connected clients.

See Also

[Read Method](#), [ReadLine Method](#), [Write Method](#), [WriteLine Method](#)

Cancel Method

Cancel a blocking socket operation.

Syntax

object.Cancel(*Handle*)

Parameters

Handle

An integer value that specifies the handle to the client session.

Return Value

None.

Remarks

The **Cancel** method cancels any blocking network operation for the specified client session. This method sets an internal flag that is periodically checked during a blocking operation, such as waiting for more data to arrive. If the client is not blocked at the time that this method is called, it will have no effect.

See Also

[Reset Method](#)

Disconnect Method

Disconnect the specified client session from the server.

Syntax

object.Disconnect(*Handle*)

Parameters

Handle

An integer value that specifies the handle to the client session.

Return Value

A value of zero is returned if the connection was terminated successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

This method terminates the specified client connection, releasing the socket handle that was allocated for the session. It is only necessary to use this method if you want the server to explicitly terminate a client connection. Normally the client will close its connection to the server, the **OnDisconnect** event will fire and the server will automatically close the socket handle allocated for that client session.

See Also

[Restart Method](#), [Stop Method](#)

FindClient Method

Return the socket handle for the client session with the specified moniker or client ID.

Syntax

object.FindClient(*Client*)

Parameters

Client

An integer value that specifies the handle to the client session or a string value that specifies a client name.

Return Value

An integer value which specifies the socket handle for the client session. If the specified moniker does not match an active client session, the method will return a value of -1 and the value of the **LastError** property will indicate the cause of the failure.

Remarks

The **FindClient** method returns a handle to the client session identified either by its moniker or client ID. The handle value that is returned can be used in conjunction with other methods that require it, such as the **Read** and **Write** methods.

If the *Client* parameter is a string, it is considered to be a client moniker and the method will search the table of connected clients and return the handle for the session that matches the specified moniker. A moniker can be assigned to the client session by setting the **ClientName** property from within an event handler such as the **OnConnect** event. Monikers are not case-sensitive, and they must be unique so that no client socket for a particular server can have the same moniker. The maximum length for a moniker is 127 characters.

If the *Client* parameter is an integer, it is considered to be a client ID and the method will return the handle for the client session that matches that ID. The ID for a client session can be obtained using the **ClientId** property from within an event handler such as the **OnConnect** event. Each client connection that is accepted by the server is assigned a unique numeric value, and unlike the socket handle for the client session, a client ID will not be reused throughout the life of the server.

See Also

[ClientCount Property](#), [ClientHandle Property](#), [ClientId Property](#), [ClientName Property](#)

Initialize Method

Initialize the control and validate the runtime license key.

Syntax

```
object.Initialize( [LicenseKey] )
```

Parameters

LicenseKey

An optional string value which specifies a runtime license key used to initialize the control. If the license key is omitted or passed as an empty string, a development license must be installed on the local system.

Return Value

A value of zero is returned if the control was initialized successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

This method dynamically loads other system libraries and allocates thread local storage. In most cases, it is not necessary to call this method directly because it is automatically invoked when an instance of the control is created by the container. However, if the control is created dynamically using **CreateObject** or a similar method, this must be the first method that is called before you attempt to modify any property values or invoke other methods. Failure to initialize the control may result in subsequent errors and/or cause an exception to be raised.

Each time the **Initialize** method is invoked, it increments an internal counter that keeps track of the number of times that it has been called by any thread in the current process. The **Uninitialize** method decrements this counter, and when the usage count drops to zero, the control will automatically unload the system libraries that it has dynamically loaded and will destroy the process heap that was allocated when the first instance of the control was created.

If the control is being used within another DLL, it is important that you do not attempt to create an instance of the control or call the **Initialize** method from within the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is written in C++ and it is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for any static and global C++ objects and has the same restrictions.

Example

```
Set objServer = CreateObject("SocketTools.InternetServer.11")

nError = objServer.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize the InternetServer control"
End
End If
```

See Also

[IsInitialized Property](#), [Uninitialize Method](#)

Peek Method

Return data read from the specified client session, but do not remove it from the socket buffer.

Syntax

object.Peek(Handle, Buffer, [Length])

Parameters

Handle

An integer value that specifies the handle to the client session.

Buffer

A buffer that the data will be stored in. If the variable is a **String** then the data will be returned as a string of characters. This is the most appropriate data type to use if the server is sending data that consists of printable characters. If the server is sending binary data, it is recommended that a **Byte** array be used instead. This parameter must be passed by reference.

Length

A numeric value which specifies the number of bytes to read. Its maximum value is $2^{31}-1 = 2147483647$. This argument is required to be present for string data. If a value is specified for this argument for other permissible types of data, and it is less than number of bytes that is determined by the control, then **Length** will override the internally computed value. If the argument is omitted, then the maximum number of bytes to read is determined by the size of the buffer.

Return Value

If the method succeeds, it will return the number of bytes available to read from the socket without causing the thread to block. A return value of zero indicates that there is no data available to read at that time. If an error occurs, a value of -1 is returned.

Remarks

The **Peek** method reads the specified number of bytes from the specified socket and copies them into the buffer, but it does not remove the data from the internal socket buffer. Note that it is possible for the returned data to contain embedded null characters.

The data returned by the **Peek** method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation, up to the specified buffer size. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. Using this method in a loop to poll a socket may cause the server application to become non-responsive. To determine if there is data available to be read, use the **IsReadable** property.

See Also

[IsReadable Property](#), [Read Method](#), [ReadLine Method](#), [Write Method](#), [WriteLine Method](#), [OnRead Event](#), [OnWrite Event](#)

Read Method

Return data read from the specified client session.

Syntax

object.Read(Handle, Buffer, [Length])

Parameters

Handle

An integer value that specifies the handle to the client session.

Buffer

A buffer that the data will be stored in. If the variable is a **String** then the data will be returned as a string of characters. This is the most appropriate data type to use if the server is sending data that consists of printable characters. If the server is sending binary data, a **Byte** array should be used instead. This parameter must be passed by reference.

Length

A numeric value which specifies the number of bytes to read. Its maximum value is $2^{31}-1 = 2147483647$. This argument is required to be present for string data. If a value is specified for this argument for other permissible types of data, and it is less than number of bytes that is determined by the control, then **Length** will override the internally computed value. If the argument is omitted, then the maximum number of bytes to read is determined by the size of the buffer.

Return Value

The number of bytes actually read from the socket is returned by this method. If an error occurs, a value of -1 is returned.

Remarks

The **Read** method returns data that has been sent by the client to the server, up to the number of bytes specified. If no data is available to be read, the application will wait until data is returned by the server or the client connection is closed.



If the data contains binary characters, particularly non-printable control characters and embedded nulls, you should always provide a **Byte** array to the **Read** method. When you provide a **String** variable as the buffer, the control will process the data as text. Binary characters may be interpreted as 8-bit ANSI character encoding and embedded null characters will corrupt the data. Reading the data into a byte array ensures that you receive the data exactly as it was sent by the server.

See Also

[CodePage Property](#), [IsReadable Property](#), [Peek Method](#), [ReadLine Method](#), [Write Method](#), [OnRead Event](#), [OnWrite Event](#)

ReadByte Method

Read a byte of data from the client.

Syntax

object.ReadByte(*Handle*)

Parameters

Handle

An integer value that specifies the handle to the client session.

Return Value

The integer value of the byte read from the socket. If an error occurs, the method will return a value of -1 and the program should check the value of the **LastError** property to determine the specific cause of the error.

Remarks

The **ReadByte** method returns one byte of data that has been read from the client socket specified by the *Handle* argument. If no data is available to be read, an error will be generated if the control is non-blocking mode. If the control is in blocking mode, the program will stop until a byte of data is returned by the server or the connection is closed.

See Also

[IsReadable Property](#), [Timeout Property](#), [Read Method](#), [Write Method](#), [WriteByte Method](#), [OnRead Event](#)

ReadLine Method

Read up to a line of data from the socket and returns it in a string buffer.

Syntax

object.ReadLine(*Handle*, *Buffer*, [*Length*])

Parameters

Handle

An integer value that specifies the handle to the client session.

Buffer

A buffer that the data will be stored in. If the variable is a String then the data will be returned as a string of characters. This is the most appropriate data type to use if the server is sending data that consists of printable characters. If the server is sending binary data, it is recommended that a Byte array be used instead. This parameter must be passed by reference.

Length

A numeric value which specifies the number of bytes to read. Its maximum value is $2^{31}-1 = 2147483647$. This argument is required to be present for string data. If a value is specified for this argument for other permissible types of data, and it is less than number of bytes that is determined by the control, then **Length** will override the internally computed value. If the argument is omitted, then the maximum number of bytes to read is determined by the size of the buffer.

Return Value

This method will return True if a line of data has been read. If an error occurs or there is no more data available to read, then the method will return False. It is possible for data to be returned in the string buffer even if the return value is False. Applications should check the length of the string after the method returns to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the function return value.

Remarks

The **ReadLine** method reads data from the socket up to the specified number of bytes or until an end-of-line character sequence is encountered. Unlike the **Read** method which reads arbitrary bytes of data, this function is specifically designed to return a single line of text data in a string variable. When an end-of-line character sequence is encountered, the function will stop and return the data up to that point; the string will not contain the carriage-return or linefeed characters.

There are some limitations when using the **ReadLine** method. The method should only be used to read text, never binary data. In particular, it will discard nulls, linefeed and carriage return control characters. This method will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection.

The **Read** and **ReadLine** methods can be intermixed, however be aware that the **Read** method will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

See Also

[CodePage Property](#), [IsReadable Property](#), [Peek Method](#), [Read Method](#), [Write Method](#), [WriteLine](#)

Reject Method

Rejects a connection request from a remote host.

Syntax

object.Reject

Parameters

None.

Return Value

A value of zero is returned if the rejection was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Reject** method rejects a pending client connection and the remote host will see this as the connection being aborted. If there are no pending client connections at the time, this method will immediately return with an error indicating that the operation would cause the thread to block.

This method can only be used inside the **OnAccept** event, when the server accepts the pending client connection. If this method is called outside of an event handler, it will fail.

Rejecting a client connection can cause the client to terminate abnormally unless it is designed to handle aborted connection attempts. It is not recommended that you use this method unless you understand the implications of doing so. To gracefully terminate a client connection, use the **Disconnect** method.

See Also

[Abort Method](#), [Disconnect Method](#), [Start Method](#), [OnAccept Event](#)

Reset Method

Reset the internal state of the control, stopping the server and terminating all client connections.

Syntax

object.Reset

Parameters

None.

Return Value

None.

Remarks

The **Reset** method resets the internal state of the control. Property values are initialized to their internal defaults, open network connections will be closed and any handles allocated by the control will be released. If the server is active when this method is called, the method will return immediately and the server shutdown process will proceed asynchronously in the background.

If this method is used to forcibly stop an active server, no further events will be generated by the control. The **OnDisconnect** event will not fire for each client session that is terminated and the **OnStop** event will not fire when the shutdown process has completed. If your application depends on these events, you should not use the **Reset** method to stop an active server.

See Also

[Restart Method](#), [Stop Method](#), [OnStop Event](#)

Resolve Method

Resolves a host name to a host IP address.

Syntax

object.Resolve(HostName, IpAddress)

Parameters

HostName

A string value that specifies the host name to resolve.

IpAddress

A string that will contain the IP address for the specified host name when the method returns. This parameter must be passed by reference.

Return Value

A value of zero is returned if the host name could be resolved into an IP address. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

See Also

[ClientAddress Property](#), [ClientHost Property](#), [ServerAddress Property](#), [ServerName Property](#)

Restart Method

Restart the server, terminating all active client connections

Syntax

object.Restart

Parameters

None.

Return Value

A value of zero is returned if the operation was successful, otherwise a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Restart** method terminates all active client connections, recreates a new listening socket bound to the same address and port number, and then resumes accepting new client connections.

See Also

[IsActive Property](#), [IsListening Property](#), [ReuseAddress Property](#), [Start Method](#), [Stop Method](#),

Resume Method

Resume accepting new client connections.

Syntax

object.Restart

Parameters

None.

Return Value

A value of zero is returned if the operation was successful, otherwise a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Resume** method instructs the server to resume accepting new client connections. Any pending client connections that were requested while the server was suspended will be accepted.

See Also

[IsActive Property](#), [IsListening Property](#), [Restart Method](#), [Start Method](#), [Stop Method](#), [Suspend Method](#)

Start Method

Start listening for client connections on the specified IP address and port number.

Syntax

object.Start([LocalAddress], [LocalPort], [Backlog], [MaxClients], [Timeout], [Options])

Parameters

LocalAddress

An optional string value that specifies the IP address of the network adapter that the control should use when listening for connection requests. If this is an empty string, the server will listen for connection on all valid network interfaces configured for the local system. If this argument is not specified, the control will accept connections on the address specified by the value of the **ServerAddress** property.

LocalPort

An optional integer value that specifies the port number to listen for connections on. If this argument is not provided, it defaults to the value specified by the **ServerPort** property.

Backlog

An optional integer value that specifies the maximum size of the queue used to manage pending connections to the service. If the argument is set to value which exceeds the maximum size for the underlying service provider, it will be silently adjusted to the nearest legal value. On Windows workstations, the maximum backlog value is 5. On Windows servers, the maximum value is 200. If this argument is not provided, the value specified by the **Backlog** property will be used.

MaxClients

An optional integer value that specifies the maximum number of clients that may connect to the server. If this argument is not provided, the value specified by the **MaxClients** property will be used. A value of zero specifies that there is no fixed limit to the number of active client connections that may be established with the server. This value can be adjusted after the server has been created by calling the **Throttle** method

Timeout

An optional integer value that specifies the number of seconds the control will wait for a network operation to complete. If this argument is not specified, the value of the **Timeout** property will be used as the default

Options

An optional integer value that specifies specifies one or more socket options which are to be used when establishing the connection. The value is created by combining the options using a bitwise Or operator. Note that if this argument is specified, it will override any property values that are related to that option.

Value	Description
swOptionDontRoute	This option specifies default routing should not be used. This option should not be specified unless absolutely necessary.
swOptionKeepAlive	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is the same as setting the KeepAlive property to a value of True.

swOptionReuseAddress	This option specifies the local address can be reused when the server is stopped and immediately restarted. This is the same as setting the ReuseAddress property to a value of True.
swOptionNoDelay	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host. This is the same as setting the NoDelay property to a value of True.
swOptionSecure	This option specifies the server will enable the security protocols and negotiate with the client to establish an encrypted session. This is the same as setting the Secure property to a value of True.< </td>

Return Value

A value of zero is returned if the operation was successful, otherwise a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Start** method begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

See Also

[MaxClients Property](#), [ServerAddress Property](#), [ServerPort Property](#), [Timeout Property](#), [Restart Method](#), [Stop Method](#), [OnStart Event](#)

Stop Method

Stop listening for new client connections and terminate all client sessions.

Syntax

object.**Stop**

Parameters

None.

Return Value

A value of zero is returned if the operation was successful, otherwise a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Stop** method instructs the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. This method will block waiting for the clients to disconnect and the server thread to terminate. Once the server has stopped, the **OnStop** event will fire.

Clients that are disconnected using the **Stop** method are terminated immediately and will not generate an **OnDisconnect** event. If your application is using this event to perform some cleanup on a per-client basis, then you should shutdown the server by first calling the **Suspend** method to prevent new connections from being accepted and terminate all active client sessions. The **OnDisconnect** event will fire for each client as it disconnects from the server, and when the last client has disconnected, the **OnIdle** event will fire. You can then call the **Stop** method to complete the shutdown of the server.

After the server has been stopped, the closed listening socket will go into a TIME-WAIT state which prevents an application from reusing the same address and port number bound to that socket for a brief period of time, typically two to four minutes. This is normal behavior designed to prevent delayed or misrouted packets of data from being read by a subsequent connection. To immediately start a new server using the same local address and port number, set the **ReuseAddress** property to a value of True.

See Also

[IsActive Property](#), [Restart Method](#), [Start Method](#), [Suspend Method](#), [OnStop Event](#)

Suspend Method

Suspend accepting new client connections.

Syntax

object.Suspend

Parameters

None.

Return Value

A value of zero is returned if the operation was successful, otherwise a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Suspend** method instructs the server to suspend accepting new client connections. Any incoming client connections will be queued up to the maximum backlog value specified when the server was started. To resume accepting client connections, call the **Resume** method.

It is not recommended that you leave a server in a suspended state for extended periods of time. Once the connection backlog queue has filled, subsequent incoming client connections will be rejected.

See Also

[IsActive Property](#), [IsListening Property](#), [Restart Method](#), [Resume Method](#), [Start Method](#), [Stop Method](#)

Throttle Method

Limit the maximum number of client connections, connections per IP address and connection rate.

Syntax

object.**Throttle**([*MaxClients*], [*MaxClientsPerAddress*], [*ConnectionRate*])

Parameters

MaxClients

An optional integer value that specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections.

MaxClientsPerAddress

An optional integer value that specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is no limit on the number of client connections per address.

ConnectionRate

An optional integer value that specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate.

Return Value

A value of zero is returned if the operation was successful, otherwise a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Throttle** method limits the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **Start** method is called with the maximum number of clients set to 100, and then the **Throttle** method is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the *ConnectionRate* value will force the server to slow down the rate at which it will accept incoming client connection requests. For example, setting this parameter to a value of 1000 would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being rejected.

See Also

[MaxClients Property](#), [Timeout Property](#), [Start Method](#), [Stop Method](#),

Uninitialize Method

Uninitialize the control and release any system resources that were allocated.

Syntax

object.Uninitialize

Parameters

None.

Return Value

None.

Remarks

The **Uninitialize** method terminates any connection established by the control and resets the internal state of the control. Any active client sessions will be terminated and the server will stop listening for new client connections. Any items in the server FIFO queue will be removed and the memory allocated for the queue will be released. This method is not typically used because any resources that have been allocated by an instance of the control will automatically be released when it is destroyed.

Each time the **Initialize** method is invoked, it increments an internal counter that keeps track of the number of times that it has been called by any thread in the current process. The **Uninitialize** method decrements this counter, and when the usage count drops to zero, the control will automatically unload the system libraries that it has dynamically loaded and will destroy the process heap that was allocated when the first instance of the control was created. An application should only call the **Uninitialize** method if it has explicitly called the **Initialize** method.

See Also

[Initialize Method](#)

Write Method

Write data to the specified client session.

Syntax

object.Write(*Handle*, *Buffer*, [*Length*])

Parameters

Handle

An integer value that specifies the handle to the client session.

Buffer

A buffer variable that contains the data to be written to the server. If the variable is a **String** type, then the data will be written as a string of characters. This is the most appropriate data type to use if the server expects text data that consists of printable characters. If the server is expecting binary data, it is recommended that a **Byte** array be used instead.

Length

A numeric value which specifies the number of bytes to write. Its maximum value is $2^{31}-1 = 2147483647$. If a value is specified for this argument and it is greater than the actual size of the buffer, then the **Length** argument will be ignored and the entire contents of the buffer will be written. If the argument is omitted, then the maximum number of bytes to write is determined by the size of the buffer.

Return Value

This method returns the number of bytes actually written to the socket, or -1 if an error was encountered.

Remarks

The **Write** method sends the data in **Buffer** to the specified client socket. Typically the data is copied to an internal buffer and control is immediately returned to the calling thread. If the buffer is full, the current thread will block until the data can be sent. If the client does not acknowledge the data that is being sent to it, this method will eventually fail with an error indicating that the connection has been aborted.



If the data contains binary characters, particularly non-printable control characters and embedded nulls, you should always provide a **Byte** array to the **Write** method. When you provide a **String** variable as the buffer, the control will process the data as text. If the string contains Unicode characters, it will automatically be converted to 8-bit encoded text prior to being written. Using a byte array ensures that binary data will be sent as-is without being encoded.

See Also

[CodePage Property](#), [IsWritable Property](#), [Timeout Property](#), [Read Method](#), [Write Method](#), [OnWrite Event](#)

WriteByte Method

Write a byte of data to the client.

Syntax

object.WriteByte(*Handle*, *Value*)

Parameters

Handle

An integer value that specifies the handle to the client session.

Value

An integer or character value that specifies the byte of data that should be sent to the remote host. If the argument is a numeric value, it will be automatically converted to its equivalent byte value and written to the socket. If the argument is a string value, then the first character will be written to the socket.

Return Value

This method returns a boolean value. If the byte of data was successfully written to the socket, the method will return True. If the data could not be written to the socket, the method will return False and the application should check the value of the **LastError** property to determine the exact cause of the failure.

Remarks

The **WriteByte** method writes a single byte of data to the client socket specified by the *Handle* argument. Typically the data is copied to an internal buffer and control is immediately returned to the calling thread. If the buffer is full, the current thread will block until the data can be sent. If the client does not acknowledge the data that is being sent to it, this method will eventually fail with an error indicating that the connection has been aborted.

See Also

[IsWritable Property](#), [Timeout Property](#), [Read Method](#), [ReadByte Method](#), [Write Method](#), [OnWrite Event](#)

WriteLine Method

Write a line of data to the specified client session, terminated with a carriage-return and linefeed.

Syntax

object.WriteLine(*Handle*, [*Buffer*])

Parameters

Handle

An integer value that specifies the handle to the client session.

Buffer

An optional string which contains the data that will be sent to the remote host. The data will always be terminated with a carriage-return and linefeed control character sequence. If this argument is omitted, then only a carriage-return and linefeed are written to the socket. Note that if the string contains a null character, any data that follows the null character will be discarded.

Return Value

This method returns True if the contents of the string have been written to the socket. If an error occurs, the method will return False.

Remarks

The **WriteLine** method writes a line of text to the remote host and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **Write** method which writes arbitrary bytes of data to the socket, this method is specifically designed to write a single line of text data from a string.

The **WriteLine** method should only be used to send text, never binary data. In particular, the method will discard any data that follows a null character and will append linefeed and carriage return control characters to the data stream. Calling this method will force the thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection.

The **Write** and **WriteLine** function calls can be safely intermixed.

See Also

[CodePage Property](#), [IsWritable Property](#), [Timeout Property](#), [Read Method](#), [ReadLine Method](#), [Write Method](#)

Internet Server Control Events

Event	Description
OnAccept	This event occurs when a client connects to the server
OnCancel	This event occurs when a blocking network operation is canceled
OnConnect	This event occurs when a client connection is established
OnDisconnect	This event occurs when a client connection is terminated
OnError	This event occurs when an error occurs
OnIdle	This event occurs after the last client has disconnected from the server
OnRead	This event occurs when a client has sent data to the server
OnStart	This event occurs when the server has started listening for connections
OnStop	This event occurs when the server has stopped
OnTimeout	This event occurs when a network operation times out
OnWrite	This event occurs when data can be written to the client

OnAccept Event

The **OnAccept** event is generated when a remote host connects to the server.

Syntax

Sub *object_OnAccept* ([*Index As Integer*,] **ByVal** *Handle As Variant*)

Remarks

This event occurs when a client attempts to establish a connection with the server.

The **Handle** argument specifies the socket descriptor of the server that has accepted the connection. The **ClientAddress** property may be used to determine the IP address of the client. To prevent the client from completing the connection, call the **Reject** method.

After the client connection has been established and the worker thread for that client session has started, the **OnConnect** event will fire.

See Also

[ClientAddress Property](#), [Reject Method](#), [OnConnect Event](#)

OnCancel Event

The **OnCancel** event is generated when a blocking operation is canceled.

Syntax

Sub *object_OnCancel* ([*Index As Integer*,] **ByVal** *Handle As Variant*)

Remarks

This event occurs when a blocking operation on the socket, such as sending or receiving data, is canceled with the **Cancel** method. The *Handle* argument specifies the handle to the active client socket.

See Also

[Cancel Method](#), [OnError Event](#)

OnConnect Event

The **OnConnect** event is generated when a client connection is established.

Syntax

Sub *object_OnConnect* ([*Index As Integer*,] **ByVal** *Handle As Variant*)

Remarks

The **OnConnect** event is generated when the client connection to the server has completed. The **Handle** argument specifies the handle to the client socket that was allocated for the session. This handle can be used with methods such as **Read** and **Write** to exchange information with the client.

The **ClientAddress** property can be used to determine the IP address of the client which established the connection. To terminate the client connection, use the **Disconnect** method.

See Also

[ClientAddress Property](#), [Disconnect Method](#), [Read Method](#), [ReadLine Method](#), [Write Method](#), [WriteLine Method](#), [OnAccept Event](#), [OnDisconnect Event](#)

OnDisconnect Event

The **OnDisconnect** event is generated when a client connection is terminated.

Syntax

Sub *object_OnDisconnect* ([*Index As Integer*,] *ByVal Handle As Variant*)

Remarks

The **OnDisconnect** event is generated when the connection is terminated by the client and there is no more data available to be read. The ***Handle*** argument specifies the socket handle of the client session which has terminated. It is important to note that the client handle is provided for informational purposes only and the application should not attempt to read or write data using this handle. When this event fires, the connection to the client has already been closed and the handle is no longer valid.

It is not necessary to call the **Disconnect** method inside the **OnDisconnect** event handler because the connection has already been closed.

See Also

[OnConnect Event](#)

OnError Event

The **OnError** event is generated when a control error occurs.

Syntax

```
Sub object_OnError ( [Index As Integer,] ByVal Handle As Variant, ByVal ErrorCode As Variant,  
ByVal Description As Variant )
```

Remarks

This event occurs when an error occurs during a control action. Visual Basic errors do not generate this event.

The ***Handle*** argument specifies the handle to the server or the specific client session which is associated with the error.

The ***ErrorCode*** argument specifies the last error that has occurred. If the error is network related, the error code values returned by the control correspond to those returned by the standard Windows Sockets library.

The ***Description*** argument is a string that describes the error.

See Also

[LastError Property](#), [LastErrorString Property](#), [ThrowError Property](#)

OnIdle Event

The **OnIdle** event is generated after the last client has disconnected from the server.

Syntax

Sub *object_OnIdle* ([*Index As Integer*])

Remarks

This event will only occur after at least one client has connected to the server and then closes its connection or is disconnected. This event will not occur immediately after the server has started using the **Start** method, and will not occur when the server is stopped using the **Stop** method. Your application should implement an **OnStart** event handler for when the server first starts, and an **OnStop** event handler for when the server is stopped.

If one or more new client connections are accepted after this event occurs, the event will be generated again when those clients disconnect and the active client count drops to zero. Therefore it is to be expected that this event will occur multiple times over the lifetime of the server as it continues to listen for connections.

See Also

[IsActive Property](#), [Restart Method](#), [Start Method](#), [Stop Method](#), [OnStop Event](#)

OnRead Event

The **OnRead** event is generated when a client has sent data to the server.

Syntax

Sub *object_OnRead* ([*Index As Integer*,] *ByVal Handle As Variant*)

Remarks

The **OnRead** event is generated when the client sends data to the server. The *Handle* argument specifies the handle to the client socket which can be used with the **Read** or **ReadLine** methods to read the data that was sent.

The application should not call the **Read** method repeatedly inside the **OnRead** event handler. When this event fires, it guarantees that data can be read from the specified client without causing the program to enter a blocked state. However, calling this method multiple times inside the event handler may cause the application to block when there is no more data available to read and this can negatively impact the overall performance of the server.

The preferred approach is to call the **Read** method once inside the event handler, buffer and/or process the data received from the client and exit the event handler. If there is more data available to be read from the client, the **OnRead** event will fire again. If you must call the **Read** method multiple times within the event handler, first check the value of the **IsReadable** property to determine if there is data available to be read.

See Also

[IsReadable Property](#), [Peek Method](#), [Read Method](#), [Write Method](#), [OnWrite Event](#)

OnStart Event

The **OnStart** event is generated when the server starts listening for connections.

Syntax

Sub *object_OnStart* ([*Index As Integer*])

Remarks

This event occurs after the **Start** method has been called and the server and begins listening for connections from clients. An application can use this event to update the user interface and perform any additional initialization functions that are required by the application.

See Also

[IsActive Property](#), [Start Method](#), [Stop Method](#), [OnStop Event](#)

OnStop Event

The **OnStop** event is generated when the server has stopped.

Syntax

Sub *object_OnStop* ([*Index As Integer*])

Remarks

This event occurs after the **Stop** method has been called and all active client sessions have terminated. An application can use this event to update the user interface and perform any additional cleanup functions that are required by the application. If the server has a large number of active clients, this event may not occur immediately. The **OnDisconnect** event will fire for each client as the server is in the process of shutting down. During the shutdown process, the server is still considered to be active, however it will not accept any further connections. When the **OnStop** event is fired, the server thread has terminated and the listening socket has been closed.

This event will not occur if the server is forcibly stopped using the **Reset** method, or when the **Uninitialize** method is called prior to disposing an instance of the control. Applications that depend on this event should ensure that the server is shutdown gracefully using the **Stop** method prior to terminating the application.

See Also

[IsActive Property](#), [Start Method](#), [Stop Method](#), [OnDisconnect Event](#), [OnStart Event](#)

OnTimeout Event

The **OnTimeout** event is fired when a network operation times out.

Syntax

Sub *object_OnTimeout* ([*Index As Integer*,] **ByVal** *Handle As Variant*)

Remarks

The **OnTimeout** event is generated when a network operation, such as sending or receiving data, times out. The **Handle** property specifies the socket handle for the current client session when the timeout occurred.

The value of the **Timeout** property determines how long the control will wait for a network operation to complete.

See Also

[Timeout Property](#), [OnCancel Event](#)

OnWrite Event

The **OnWrite** event is generated when data can be written to the client.

Syntax

Sub *object_OnWrite* ([*Index As Integer*,] **ByVal** *Handle As Variant*)

Remarks

The **OnWrite** event is generated when the client can accept data from the server. The *Handle* argument specifies the handle to the client socket and can be used in conjunction with the **Write** or **WriteLine** methods.

This event is typically fired once when the client connection is established with the server, the session thread starts and the client socket enters a writable state. If the internal send buffer for the client socket becomes full, this event will fire again when more data can be written to the socket. It is important to note that this event is level-triggered and will not fire repeatedly if the client socket is writable. Under most circumstances this event fire only once for each client session after the initial connection has been established.

See Also

[IsWritable Property](#), [Write Method](#), [WriteLine Method](#), [OnRead Event](#)

Internet Server Control Error Codes

Value	Constant	Description
10001	swErrorNotHandleOwner	Handle not owned by the current thread
10002	swErrorFileNotFound	The specified file or directory does not exist
10003	swErrorFileNotCreated	The specified file could not be created
10004	swErrorOperationCanceled	The blocking operation has been canceled
10005	swErrorInvalidFileType	The specified file is a block or character device, not a regular file
10006	swErrorInvalidDevice	The specified device or address does not exist
10007	swErrorTooManyParameters	The maximum number of function parameters has been exceeded
10008	swErrorInvalidFileName	The specified file name contains invalid characters or is too long
10009	swErrorInvalidFileHandle	Invalid file handle passed to function
10010	swErrorFileReadFailed	Unable to read data from the specified file
10011	swErrorFileWriteFailed	Unable to write data to the specified file
10012	swErrorOutOfMemory	Out of memory
10013	swErrorAccessDenied	Access denied
10014	swErrorInvalidParameter	Invalid argument passed to function
10015	swErrorClipboardUnavailable	The system clipboard is currently unavailable
10016	swErrorClipboardEmpty	The system clipboard is empty or does not contain any text data
10017	swErrorFileEmpty	The specified file does not contain any data
10018	swErrorFileExists	The specified file already exists
10019	swErrorEndOfFile	End of file
10020	swErrorDeviceNotFound	The specified device could not be found
10021	swErrorDirectoryNotFound	The specified directory could not be found
10022	swErrorInvalidBuffer	Invalid memory address passed to function
10024	swErrorNoHandles	No more handles available to this process
10035	swErrorOperationWouldBlock	The specified operation would block the current thread
10036	swErrorOperationInProgress	A blocking operation is currently in progress
10037	swErrorAlreadyInProgress	The specified operation is already in progress
10038	swErrorInvalidHandle	Invalid handle passed to function
10039	swErrorInvalidAddress	Invalid network address specified
10040	swErrorInvalidSize	Datagram is too large to fit in specified buffer
10041	swErrorInvalidProtocol	Invalid network protocol specified
10042	swErrorProtocolNotAvailable	The specified network protocol is not available
10043	swErrorProtocolNotSupported	The specified protocol is not supported

10044	swErrorSocketNotSupported	The specified socket type is not supported
10045	swErrorInvalidOption	The specified option is invalid
10046	swErrorProtocolFamily	The specified protocol family is not supported
10047	swErrorProtocolAddress	The specified address is invalid for this protocol family
10048	swErrorAddressInUse	The specified address is in use by another process
10049	swErrorAddressUnavailable	The specified address cannot be assigned
10050	swErrorNetworkUnavailable	The networking subsystem is unavailable
10051	swErrorNetworkUnreachable	The specified network is unreachable
10052	swErrorNetworkReset	Network dropped connection on reset
10053	swErrorConnectionAborted	Connection was aborted due to timeout or other failure
10054	swErrorConnectionReset	Connection was reset by remote network
10055	swErrorOutOfBuffers	No buffer space is available
10056	swErrorAlreadyConnected	Connection already established with remote host
10057	swErrorNotConnected	No connection established with remote host
10058	swErrorConnectionShutdown	Unable to send or receive data after connection shutdown
10060	swErrorOperationTimeout	The specified operation has timed out
10061	swErrorConnectionRefused	The connection has been refused by the remote host
10064	swErrorHostUnavailable	The specified host is unavailable
10065	swErrorHostUnreachable	The specified host is unreachable
10067	swErrorTooManyProcesses	Too many processes are using the networking subsystem
10091	swErrorNetworkNotReady	Network subsystem is not ready for communication
10092	swErrorInvalidVersion	This version of the operating system is not supported
10093	swErrorNetworkNotInitialized	The networking subsystem has not been initialized
10101	swErrorRemoteShutdown	The remote host has initiated a graceful shutdown sequence
11001	swErrorInvalidHostName	The specified hostname is invalid or could not be resolved
11002	swErrorHostNameNotFound	The specified hostname could not be found
11003	swErrorHostNameRefused	Unable to resolve hostname, request refused
11004	swErrorHostNameNotResolved	Unable to resolve hostname, no address for specified host
12001	swErrorInvalidLicense	The license for this product is invalid
12002	swErrorProductNotLicensed	This product is not licensed to perform this operation
12003	swErrorNotImplemented	This function has not been implemented on this platform
12004	swErrorUnknownLocalHost	Unable to determine local host name
12005	swErrorInvalidHostAddress	Invalid host address specified
12006	swErrorInvalidServicePort	Invalid service port number specified

12007	swErrorInvalidServiceName	Invalid or unknown service name specified
12008	swErrorInvalidEventId	Invalid event identifier specified
12009	swErrorOperationNotBlocking	No blocking operation in progress on this socket
12101	swErrorSecurityNotInitialized	Unable to initialize security interface for this process
12102	swErrorSecurityContext	Unable to establish security context for this session
12103	swErrorSecurityCredentials	Unable to open client certificate store or establish client credentials
12104	swErrorSecurityCertificate	Unable to validate the certificate chain for this session
12105	swErrorSecurityDecryption	Unable to decrypt data stream
12106	swErrorSecurityEncryption	Unable to encrypt data stream
12201	swErrorOperationNotSupported	The specified operation is not supported
12330	swErrorFeatureNotSupported	The specified feature is not supported
12337	swErrorMaximumConnections	The maximum number of client connections exceeded
12338	swErrorThreadCreationFailed	Unable to create a new thread for the current process
12339	swErrorInvalidThreadHandle	The specified thread handle is no longer valid
12340	swErrorThreadTerminated	The specified thread has been terminated
12341	swErrorThreadDeadlock	The operation would result in the current thread becoming deadlocked
12342	swErrorInvalidClientMoniker	The specified moniker is not associated with any client session
12343	swErrorClientMonikerExists	The specified moniker has been assigned to another client session
12344	swErrorServerInactive	The specified server is not listening for client connections
12345	swErrorServerSuspended	The specified server is suspended and not accepting client connections

SocketWrench ActiveX Control

A general purpose TCP/IP networking component for developing client and server applications.

Reference

- [Properties](#)
- [Methods](#)
- [Events](#)
- [Error Codes](#)

Control Information

Object Name	SocketWrenchCtl.SocketWrench
File Name	CSWSKX12.OCX
Version	12.0.3024.1286
ProgID	SocketTools.SocketWrench.12
ClassID	FA174594-F986-428D-8E8B-25E7C61A16D1
Threading Model	Apartment
Help File	CSW12HLP.CHM
Dependencies	None
Standards	RFC 768, RFC 791, RFC 793

Overview

The SocketWrench control provides a simplified interface for the standard Windows Sockets API used to develop Internet and intranet applications using the TCP/IP protocol. With SocketWrench, you can create both client and server applications, as well as send and receive UDP datagrams. SocketWrench also supports secure connections using the standard Transport Layer Security (TLS) protocol. Enabling the security features of the control is done by setting a single property, and all of the data that is exchanged between your application and the remote host will be encrypted.

Instead of using complex API calls, virtually all network functions can be performed by setting the control's properties and responding to events. For developers who are not familiar with the details of Internet programming, SocketWrench can also insulate them from many of the common pitfalls, without sacrificing functionality or flexibility.

Each instance of the control that you use corresponds to one socket which may or may not be currently connected to a remote host. If you need access to multiple sockets, you simply create multiple instances of the control. This is most commonly needed when your application acts a server and must be able to handle several connections at one time.

Requirements

The SocketWrench control is self-registering and compatible with any programming language that supports the Component Object Model (COM) and the ActiveX control specification. If you are using Visual Basic 6.0, Service Pack 6 (SP6) is required. We recommend installing all available updates for your development tools. Although the Visual Basic 6.0 runtime is supported on Windows 11, the IDE itself is no longer officially supported by Microsoft. Legacy IDEs and compilers may also require

elevated (administrator) privileges to run properly.

The control is supported on Windows 7, Windows Server 2008 R2, and later versions of both desktop and server editions of Windows. For Windows 7, Service Pack 1 (SP1) is the minimum requirement. For Windows 8 systems, the 8.1 update must be installed. On ARM-based platforms, the minimum supported version is Windows 10 build 1709.

SocketWrench includes components for both 32-bit (x86) and 64-bit (x64) Windows platforms. All modern versions of Windows are 64-bit, but we continue to include 32-bit components for compatibility with legacy development tools such as Visual Basic 6.0. We also provide native ARM64 ActiveX controls for systems running Windows on ARM. Note that 32-bit applications on ARM run using x86 emulation, and Microsoft no longer supports native ARM32 applications.

If you are using VBScript, Visual Basic for Applications (VBA), or another scripting environment on an ARM-based system, be aware that 64-bit hosts will require ARM64 components, while 32-bit hosts will use x86 components under emulation. This includes applications created with Visual Basic 6.0. To reduce deployment complexity and avoid confusion, SocketWrench does not include ARM32 ActiveX controls in its standard redistribution packages but they are available on request for legacy or specialized scenarios.

Distribution

When you distribute an application that uses this control, you can either install the file in the same folder as your application executable or as a shared component in the appropriate system folder. If you install the control in the system folder, it is important that you distribute the correct version for the target platform and it must be registered. If you install the control in the same folder as your executable, it is recommended that you use registration-free activation or COM redirection to ensure that the correct version of the control is loaded by the application.

SocketWrench Control Properties

Property	Description
AdapterAddress	Returns the IP address associated with the specified network adapter
AdapterCount	Returns the number of available local and remote network adapters
AtMark	A read-only property that returns True if the next receive will return urgent data
AutoResolve	Determines if host names and addresses are automatically resolved
Backlog	Gets and sets the number of client connections that may be queued by a listening socket
Blocking	Gets and sets the blocking state of the control
Broadcast	Determines if datagrams should be broadcast over the network
ByteOrder	Gets and sets the byte order in which integer data will be written to and read from the socket
CertificateExpires	Return the date and time that the server certificate expires
CertificateIssued	Return the date and time that the server certificate was issued
CertificateIssuer	Returns information about the organization that issued the server certificate
CertificateName	Gets and sets the common name for the security certificate
CertificatePassword	Gets and sets the password associated with the certificate
CertificateStatus	Return the status of the server certificate
CertificateStore	Gets and sets the name of the certificate store or file
CertificateSubject	Returns information about the organization to which the server certificate was issued
CertificateUser	Gets and sets the user that owns the client certificate
CipherStrength	Return the length of the key used by the encryption algorithm
CodePage	Gets and sets the code page used when reading and writing text
ExternalAddress	Return the external IP address assigned to the local system
HashStrength	Return the length of the message digest that was selected
HostAddress	Gets and sets the IP address of the remote host
HostAlias	Returns the aliases defined for the current hostname
HostFile	Gets and sets the name of an alternate host file
HostName	Gets and sets the name of the remote host
InLine	Sets or returns if urgent data is received in-line with non-urgent data
Interval	Gets and sets the number of milliseconds between calls to the control's timer event
IsBlocked	Determine if the control is blocked performing an operation
IsClosed	Determine if the connection has been closed by the remote host
IsConnected	Determine if the control is connected to a remote host
IsInitialized	Determine if the control has been initialized
IsListening	Returns if the socket is listening for connections
IsReadable	Determine if data can be read from the socket without blocking

IsWritable	Determine if data can be written to the socket without blocking
KeepAlive	Set or return if keep-alive packets are sent on a connected socket
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error that occurred
Linger	Gets and sets the number of seconds to wait for the socket to close
LocalAddress	Return the IP address of the local host
LocalName	Return the name of the local host
LocalPort	Gets and sets the port number for a local listening socket
NoDelay	Enable or disable the Nagle algorithm
Options	Gets and sets the options that are used in establishing a connection
PeerAddress	Return the IP address of the remote peer
PeerName	Return the name of the remote peer
PeerPort	Return the port number of the remote connection or datagram
PhysicalAddress	Return the MAC address for the local host's Ethernet or Token Ring adapter
Protocol	Gets and sets the protocol that should be used to create the socket
RemotePort	Gets and sets the port number for a remote connection
ReservedPort	Set or return if a reserved local port number should be allocated
ReuseAddress	Set or return if an address can be reused
Secure	Set or return if a connection to the remote host is secure
SecureCipher	Return the encryption algorithm used to establish a secure connection
SecureHash	Return the message digest selected when establishing a secure connection
SecureKeyExchange	Return the key exchange algorithm used to establish a secure connection
SecureProtocol	Gets and sets the security protocol used to establish a secure connection
ThrowError	Enables or disables error handling by the application
Timeout	Gets and sets the amount of time until a blocking operation fails
Trace	Enable or disable socket function level tracing
TraceFile	Specify the socket function trace output file
TraceFlags	Gets and sets the socket function tracing flags
Urgent	Send or receive urgent data
Version	Return the current version of the control
VpnAdapter	Return the name of the active VPN adapter.
VpnAddress	Return the local IP address assigned to the active VPN adapter.

AdapterAddress Property

Returns the IP address associated with the specified network adapter.

Syntax

object.AdapterAddress(Index)

Remarks

The **AdapterAddress** property array returns the IP addresses that are associated with the local network or remote dial-up network adapters configured on the system. The **AdapterCount** property can be used to determine the number of adapters that are available.

Multihomed systems with more than one local network adapter, or a combination of local and dial-up adapters will not be listed in a specific order. An application should not make the assumption that the address returned by **AdapterAddress(0)** always refers to a local network adapter.

Note that it is possible that the **AdapterCount** property will return 0, and **AdapterAddress(0)** will return an empty string. This indicates that the system does not have a physical network adapter with an assigned IP address, and there are no dial-up networking connections currently active. If a dial-up networking connection is established at some later point, the **AdapterCount** property will change to 1, and the **AdapterAddress(0)** property will return the IP address allocated for that connection.

When using Visual Studio .NET, you must use the accessor method **get_AdapterAddress** instead of the property name, otherwise an error will be returned indicating that it not a member of the control class.

Data Type

String

See Also

[AdapterCount Property](#), [LocalAddress Property](#), [LocalName Property](#), [PhysicalAddress Property](#)

AdapterCount Property

Returns the number of available local and remote network adapters.

Syntax

object.AdapterCount

Remarks

The **AdapterCount** property returns the number of local and remote dial-up networking adapters available on the local system. This value can be used in conjunction with the **AdapterAddress** property array to enumerate the IP addresses assigned to the various network adapters.

Note that it is possible that the **AdapterCount** property will return 0, and **AdapterAddress**(0) will return an empty string. This indicates that the system does not have a physical network adapter with an assigned IP address, and there are no dial-up networking connections currently active. If a dial-up networking connection is established at some later point, the **AdapterCount** property will change to 1, and the **AdapterAddress**(0) property will return IP address allocated for that connection.

Data Type

Integer (Int32)

See Also

[AdapterAddress Property](#), [LocalAddress Property](#), [LocalName Property](#)

AtMark Property

A read-only property that returns True if the next receive will return urgent data.

Syntax

object.AtMark

Remarks

This property can only be used if the Protocol is swProtocolTcp and the InLine property has been set to True. Reading this property is the same as using the SIOCATMARK option with the **ioctlsocket** function.

Data Type

Boolean

See Also

[Urgent Property](#), [OnRead Event](#)

AutoResolve Property

Determines if host names and addresses are automatically resolved.

Syntax

object.AutoResolve [= { True | False }]

Remarks

Setting the **AutoResolve** property determines if the control automatically resolves host names and addresses specified by the **HostName** and **HostAddress** properties. If set to True, setting the **HostName** property will cause the control to automatically determine the corresponding IP address and set the **HostAddress** property accordingly. Likewise, setting the **HostAddress** property will cause the control to determine the host name and set the **HostName** property. Setting the property to False prevents the control from resolving host names until a connection attempt is made.

Note: When using the domain name service (DNS), setting the **HostName** or **HostAddress** property may cause the thread to block, sometimes for several seconds, until the name or address is resolved. To prevent this behavior, set **AutoResolve** to False.

Data Type

Boolean

See Also

[HostAddress Property](#), [HostFile Property](#), [HostName Property](#), [Resolve Method](#)

Backlog Property

Gets and sets the number of client connections that may be queued by a listening socket.

Syntax

object.Backlog [= *backlog*]

Remarks

The **Backlog** property specifies the maximum size of the queue used to manage pending connections to the server. If the property is set to value which exceeds the maximum size for the underlying service provider, it will be silently adjusted to the nearest legal value. There is no standard way to determine what the maximum backlog value is.

This property must be set to the desired value before the **Listen** method is called, if the **Listen** method is used with default parameters. The default backlog value is 5 on all Windows platforms. The Windows Server platforms support a maximum backlog value of 200.

Note that this property does not specify the total number of connections that the server application may accept. It only specifies the size of the backlog queue which is used to manage pending client connections. Once the client connection has been accepted, it is removed from the queue.

Data Type

Integer (Int32)

See Also

[IsListening Property](#), [OnAccept Event](#), [Accept Method](#), [Listen Method](#)

Blocking Property

Gets and sets the blocking state of the control.

Syntax

object.**Blocking** [= { True | False }]

Remarks

Setting the **Blocking** property determines if control actions complete synchronously or asynchronously. If set to True, then each control action (such as sending or receiving data) will return when the operation has completed or timed-out. If set to False, control actions will return immediately. If the operation would result in the control blocking (such as attempting to receive data when none has been written), an error is generated. Control events such as **OnDisconnect**, **OnRead** and **OnWrite** are only fired if the socket is non-blocking.

Data Type

Boolean

See Also

[IsBlocked Property](#)

Broadcast Property

Determines if datagrams should be broadcast over the network.

Syntax

object.**Broadcast** [= { True | False }]

Remarks

If the **Broadcast** property is set to a value of true, the datagram written to the socket will be broadcast to all systems on the network. Use of this property is restricted to the swProtocolUdp protocol.

Data Type

Boolean

See Also

[InLine Property](#), [KeepAlive Property](#), [ReuseAddress Property](#), [Route Property](#), [Protocol Property](#)

ByteOrder Property

Gets and sets the byte order in which integer data will be written to and read from the socket.

Syntax

object.ByteOrder [= 0 | 1]

Remarks

The **ByteOrder** property is used to specify how 16-bit (short) integer and 32-bit (long) integer data is written to and read from the socket. The default value for this property is 0, which specifies that integers should be written in the native byte order for the local machine. A value of 1 indicates that integers should be written in network byte order.

When applications write integer values on a socket (instead of string representations of those values), they should typically be converted to network byte order before they are sent. Likewise, when an integer value is read, it should then be converted from the network byte order back to the byte order used by the local machine. The native byte order, also called the host byte order, should only be used if it can be assured that both the sender and the receiver are running on an identical or compatible machine architectures (for example, if both systems are Intel-based).

This property will affect how data is read by the **Read** method and by the **Write** method, if the Variant data that is being read or written is recognized as integer data.

Data Type

Integer (Int32)

CertificateExpires Property

Return the date and time that the server certificate expires.

Syntax

object.CertificateExpires

Remarks

The **CertificateExpires** property returns the date and time that the server certificate expires. This property will return an empty string if a secure connection has not been established with the server.

Data Type

String

See Also

[CertificateIssued Property](#), [CertificateIssuer Property](#), [CertificateStatus Property](#), [CertificateSubject Property](#), [Secure Property](#)

CertificateIssued Property

Return the date and time that the server certificate was issued.

Syntax

object.CertificateIssued

Remarks

The **CertificateIssued** property returns the date and time that the server certificate was issued. This property will return an empty string if a secure connection has not been established with the server.

Data Type

String

See Also

[CertificateExpires Property](#), [CertificateIssuer Property](#), [CertificateStatus Property](#), [CertificateSubject Property](#), [Secure Property](#)

CertificateIssuer Property

Returns information about the organization that issued the server certificate.

Syntax

object.CertificateIssuer

Remarks

The **CertificateIssuer** property returns a string that contains information about the organization that issued the server certificate. The string value is a comma separated list of tagged name and value pairs. In the nomenclature of the X.500 standard, each of these pairs are called a relative distinguished name (RDN), and when concatenated together, forms the issuer's distinguished name (DN). For example:

C=US, O="RSA Data Security, Inc.", OU=Secure Server Certification Authority

To obtain a specific value, such as the name of the issuer or the issuer's country, the application must parse the string returned by this property. Some of the common tokens used in the distinguished name are:

Name	Description
C	The ISO standard two character country code
S	The name of the state or province
L	The name of the city or locality
O	The name of the company or organization
OU	The name of the department or organizational unit
CN	The common name; with X.509 certificates, this is the domain name of the site the certificate was issued for

This property will return an empty string if a secure connection has not been established with the server.

Data Type

String

Example

The following example demonstrates how to extract the value of a relative distinguished name token:

```
Function GetCertNameValue(ByVal strValue As String, ByVal strFieldName As String)
As String
    Dim strFieldValue As String
    Dim cchValue As Integer, cchFieldName As Integer
    Dim nOffset As Integer

    GetCertNameValue = ""
    cchValue = Len(strValue)
    cchFieldName = Len(strFieldName)

    If cchValue = 0 Or cchFieldName = 0 Then
        Exit Function
    End If
```

```

nOffset = InStr(strValue, strFieldName & "=")

If nOffset > 0 Then
    '
    ' If the field name was found in the string, then
    ' remove everything to the left of the token from
    ' the string
    '
    strFieldValue = Right(strValue, cchValue - (nOffset + cchFieldName))

    '
    ' If the value is quoted, then strip off the leading
    ' quote and look for the ending quote in the string;
    ' otherwise look for the comma that marks the end of
    ' the field name/value pair
    '
    If Left(strFieldValue, 1) = Chr(34) Then
        strFieldValue = Right(strFieldValue, Len(strFieldValue) - 1)
        nOffset = InStr(strFieldValue, Chr(34))
    Else
        nOffset = InStr(strFieldValue, ",")
    End If

    '
    ' If the offset is 0, then the name/value pair is
    ' the last token in the string; otherwise, remove
    ' everything to the right of that position
    '
    If nOffset > 0 Then
        strFieldValue = Left(strFieldValue, nOffset - 1)
    End If

    GetCertNameValue = strFieldValue
End If

```

End Function

This function could then be used to return the name of the company who issued the server certificate:

```

Dim strIssuer As String
Dim strCompanyName As String

strIssuer = HttpClient1.CertificateIssuer
If Len(strIssuer) = 0 Then
    MsgBox "A secure connection has not been established"
Else
    strCompanyName = GetCertNameValue(strIssuer, "O")
    MsgBox "This certificate was issued by " & strCompanyName
End If

```

See Also

[CertificateExpires Property](#), [CertificateIssued Property](#), [CertificateStatus Property](#), [CertificateSubject Property](#), [Secure Property](#)

CertificateName Property

Gets and sets the common name for the security certificate.

Syntax

object.CertificateName [= *name*]

Remarks

This property sets the common name or friendly name of the client certificate that should be used to establish the connection with the server, or the name of the server certificate if the control is being used to create a server application. This property is used in conjunction with the **CertificateStore** property to identify the certificate that should be used to create a security context for the session.

For client applications, it is only required that you set this property value if the server requires a client certificate for authentication. If this property is not set, a client certificate will not be provided to the server. The certificate must be designated as a client certificate and have a private key associated with it, otherwise the connection attempt will fail.

For server applications, it is required that you specify a certificate name if security has been enabled by setting the **Secure** property to True. The certificate must be designated as a server certificate and have a private key associated with it, otherwise the control will be unable to accept incoming client connections.

Certificates may be installed and viewed on the local system using the Certificate Manager that is included with the Windows operating system. For more information, refer to the documentation for the Microsoft Management Console.

Data Type

String

See Also

[CertificateStore Property](#), [Secure Property](#)

CertificatePassword Property

Gets and sets the password associated with the certificate.

Syntax

object.CertificatePassword [= *password*]

Remarks

This property sets the password that should be used to access a certificate in the specified certificate store. It is only required when the **CertificateStore** property specifies a file that contains a certificate and private key in PKCS #12 format.

Data Type

String

See Also

[CertificateName Property](#), [CertificateStore Property](#), [Secure Property](#)

CertificateStatus Property

Return the status of the server certificate.

Syntax

object.CertificateStatus

Remarks

The **CertificateStatus** property returns an integer value which identifies the status of the server certificate. This property may return one of the following values:

Value	Description
swCertificateNone	No certificate information is available. A secure connection was not established with the server.
swCertificateValid	The certificate is valid.
swCertificateNoMatch	The certificate is valid, however the domain name specified in the certificate does not match the domain name of the site that the client has connected to. This is typically the case if the HostAddress property is used rather than the HostName property. It is recommended that the client examine the CertificateSubject property to determine the domain name of the site that the certificate was issued for.
swCertificateExpired	The certificate has expired and is no longer valid. The client can examine the CertificateExpires property to determine when the certificate expired.
swCertificateRevoked	The certificate has been revoked and is no longer valid. It is recommended that the client application immediately terminate the connection if this status is returned.
swCertificateUntrusted	The certificate has not been issued by a trusted authority, or the certificate is not trusted on the local host. It is recommended that the client application immediately terminate the connection if this status is returned.
swCertificateInvalid	The certificate is invalid. This typically indicates that the internal structure of the certificate is damaged. It is recommended that the client application immediately terminate the connection if this status is returned.

This property value should be checked after the connection to the server has completed, but prior to beginning a transaction. If a secure connection has not been established, this property will return a value of zero.

Data Type

Integer (Int32)

Example

The following example establishes a secure connection to a server:

```
SocketWrench1.HostName = strHostName  
SocketWrench1.Secure = True
```

```
nError = SocketWrench1.Connect()  
If nError > 0 Then
```

```
        MsgBox "Unable to connect to server " & strHostName, vbExclamation
    Exit Sub
End If

If SocketWrench1.CertificateStatus <> swCertificateValid Then
    nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
        "Are you sure you wish to continue?", vbYesNo)

    If nResult = vbNo Then
        SocketWrench1.Disconnect
        Exit Sub
    End If
End If

SocketWrench1.Disconnect
```

See Also

[CertificateExpires Property](#), [CertificateIssued Property](#), [CertificateIssuer Property](#), [CertificateSubject Property](#), [Secure Property](#)

CertificateStore Property

Gets and sets the name of the certificate store or file.

Syntax

object.CertificateStore [= *store*]

Remarks

This property sets the name of the certificate store that contains the certificate that should be used when establishing a secure connection with the server or accepting secure client connections. The certificate may either be stored in the registry or in a file. If the certificate is stored in the registry, then this property should be set to one of the following predefined values:

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. If a certificate store is not specified, this is the default value that is used.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

In most cases the certificate will be installed in the user's personal certificate store, and therefore it is not necessary to set this property value because that is the default location that will be used to search for the certificate. This property is only used if the **CertificateName** property is also set to a valid certificate name.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU" for the current user, or "HKLM" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, it will default to the certificate store for the current user.

This property may also be used to specify a file that contains the certificate. In this case, the property should specify the full path to the file and must contain both the certificate and private key in PKCS #12 format. If the file is protected by a password, the **CertificatePassword** property must also be set to specify the password.

Data Type

String

See Also

[CertificateName Property](#), [CertificatePassword Property](#), [Secure Property](#)

CertificateSubject Property

Returns information about the organization that the server certificate was issued to.

Syntax

`object.CertificateSubject`

Remarks

The **CertificateSubject** property returns a string that contains information about the organization that the server certificate was issued for. The string value is a comma separated list of tagged name and value pairs. In the nomenclature of the X.500 standard, each of these pairs are called a relative distinguished name (RDN), and when concatenated together, forms the subject's distinguished name (DN). For example:

C=US, O="RSA Data Security, Inc.", OU=Secure Server Certification Authority

To obtain a specific value, such as the name of the subject's company or country, the application must parse the string returned by this property. Some of the common tokens used in the distinguished name are:

Name	Description
C	The ISO standard two character country code
S	The name of the state or province
L	The name of the city or locality
O	The name of the company or organization
OU	The name of the department or organizational unit
CN	The common name; with X.509 certificates, this is the domain name of the site the certificate was issued for

This property will return an empty string if a secure connection has not been established with the server.

Data Type

String

Example

The following example demonstrates how to extract the value of a relative distinguished name token:

```
Function GetCertNameValue(ByVal strValue As String, ByVal strFieldName As String)
As String
    Dim strFieldValue As String
    Dim cchValue As Integer, cchFieldName As Integer
    Dim nOffset As Integer

    GetCertNameValue = ""
    cchValue = Len(strValue)
    cchFieldName = Len(strFieldName)

    If cchValue = 0 Or cchFieldName = 0 Then
        Exit Function
    End If
```



```

nOffset = InStr(strValue, strFieldName & "=")

If nOffset > 0 Then
    '
    ' If the field name was found in the string, then
    ' remove everything to the left of the token from
    ' the string
    '
    strFieldValue = Right(strValue, cchValue - (nOffset + cchFieldName))

    '
    ' If the value is quoted, then strip off the leading
    ' quote and look for the ending quote in the string;
    ' otherwise look for the comma that marks the end of
    ' the field name/value pair
    '

    If Left(strFieldValue, 1) = Chr(34) Then
        strFieldValue = Right(strFieldValue, Len(strFieldValue) - 1)
        nOffset = InStr(strFieldValue, Chr(34))
    Else
        nOffset = InStr(strFieldValue, ",")
    End If

    '
    ' If the offset is 0, then the name/value pair is
    ' the last token in the string; otherwise, remove
    ' everything to the right of that position
    '

    If nOffset > 0 Then
        strFieldValue = Left(strFieldValue, nOffset - 1)
    End If

    GetCertNameValue = strFieldValue
End If

```

End Function

This function could then be used to return the domain name that the server certificate was issued for:

```

Dim strSubject As String
Dim strDomainName As String

strSubject = HttpClient1.CertificateSubject
If Len(strSubject) = 0 Then
    MsgBox "A secure connection has not been established"
Else
    strDomainName = GetCertNameValue(strSubject, "CN")
    MsgBox "This certificate was issued for " & strDomainName
End If

```

See Also

[CertificateExpires Property](#), [CertificateIssued Property](#), [CertificateIssuer Property](#), [CertificateStatus Property](#), [Secure Property](#)

CertificateUser Property

Gets and sets the user that owns the certificate.

Syntax

object.CertificateUser [= *username*]

Remarks

This property sets the name of the user that owns the certificate that will be used to establish a secure connection with the server or accept secure client connections. If this property is not set, the certificate store for the current user will be used when searching for the certificate. If this property is used to specify another user, the process must have the appropriate permission to access the registry location that contains the client certificate. On Windows Vista and later versions of the operating system, this requires that the process run with elevated privileges.

Data Type

String

See Also

[CertificateName Property](#), [CertificateStore Property](#), [Secure Property](#)

CipherStrength Property

Return the length of the key used by the encryption algorithm.

Syntax

object.CipherStrength

Remarks

The **CipherStrength** property returns the number of bits in the key used to encrypt the secure data stream. Common values returned by this property are 128 and 256. A key length of 40-bits or 56-bits is considered to be insecure, and subject to brute force attacks. 128-bit and 256-bit keys are considered secure. If this property returns a value of 0, this means that a secure connection has not been established with the server.

Data Type

Integer (Int32)

See Also

[HashStrength Property](#), [Secure Property](#), [SecureCipher Property](#), [SecureHash Property](#), [SecureKeyExchange Property](#), [SecureProtocol Property](#)

CodePage Property

Gets and sets the code page used when reading and writing text.

Syntax

object.CodePage [= *value*]

Remarks

The **CodePage** property is an integer value which specifies how strings are encoded when data is sent or received. Any valid code page identifier may be specified. Some common values are:

Value	Description
0	Text sent and received using a string should be converted using the ANSI code page for the current locale.
1	Text sent and received using a string should be converted using the system default OEM code page. The OEM code page typically contains characters that are used by console applications and are based on character sets commonly used by MS-DOS. You should not use this code page unless you know the server is sending text which includes OEM characters.
1252	Text sent and received using a string should be converted using the Windows ANSI code page for western European languages. This code page is commonly used by legacy Windows applications for English and some other western languages. It should be noted that while this code page is similar to ISO 8859-1 character encoding, it is not identical.
28591	Text sent and received using a string should be converted using the ISO 8859-1 code page for western European languages. This code page is commonly referred to as Latin-1 and is similar to the Windows 1252 code page.
65000	Data that is sent and received using a string should be converted using UTF-7 encoding. If this code page is specified, data written to the socket will be encoded as UTF-7 encoded Unicode. All data received from the server will be converted from UTF-7. It is not recommended that you use this code page unless you know that the remote host is sending UTF-7 encoded text.
65001	Data that is sent and received using a string should be converted using UTF-8 encoding. If this code page is specified, data written to the socket will be encoded as UTF-8 encoded Unicode. All data received from the server will be converted from UTF-8 to UTF-16 Unicode. Because UTF-8 is backwards compatible with the ASCII character set, it is safe to use this encoding option when sending and receiving ASCII text.

A complete list of available [code page identifiers](#) can be found in Microsoft's documentation for the Win32 API.

All data which is exchanged over a socket is sent and received as 8-bit bytes, typically referred to as "octets" in networking terminology. However, the internal string type used by ActiveX controls are Unicode where each character is represented by 16 bits. To send and receive data using strings, these Unicode strings are converted to a stream of bytes.

By default, strings are converted to an array of bytes using the code page for the current locale, mapping the 16-bit Unicode characters to bytes. Similarly, when reading data from the socket into a

string buffer, the stream of bytes received from the remote host are converted to Unicode before they are returned to your application.

If you are exchanging text with another system and it appears to be corrupted or characters are being replaced with question marks or other symbols, it is likely the system is sending text which is using a different character encoding. Most services use UTF-8 encoding to represent non-ASCII characters and selecting the UTF-8 code page will typically resolve the issue.



Strings are only guaranteed to be safe when sending and receiving text. Using a string data type is not recommended when reading or writing binary data to a socket. If possible, you should always use a byte array as the buffer parameter for the **Read** and **Write** methods whenever you are exchanging binary data.

For backwards compatibility, the control defaults to using the code page for the current locale. This property value directly corresponds to Windows code page identifiers, and will accept any valid code page in addition to the values listed above. Setting this property to an invalid code page will result in an error.

Data Type

Integer (Int32)

See Also

[Read Method](#), [ReadLine Method](#), [ReadStream Method](#), [Write Method](#), [WriteLine Method](#), [WriteStream Method](#)

ExternalAddress Property

Return the external IP address for the local system.

Syntax

object.ExternalAddress

Remarks

The **ExternalAddress** property returns the IP address assigned to the router that connects the local host to the Internet. This is typically used by an application executing on a system in a local network that uses a router which performs Network Address Translation (NAT). In that network configuration, the **LocalAddress** property will only return the IP address for the local system on the LAN side of the network unless a connection has already been established to a remote host.

This property can be used to determine the IP address assigned to the router on the Internet side of the connection and can be particularly useful for servers running on a system behind a NAT router. Note that you should not assign the **LocalAddress** property to the value returned by the **ExternalAddress** property. If the server is running behind a NAT router, the router must be configured to forward incoming connections to the appropriate address on the local network.

It may not be possible to determine the external IP address for the local system for a number of reasons, particularly if it is behind a firewall or uses a proxy server that restricts access to external sites on the Internet. If you are using a VPN (Virtual Private Network), this property will return the external IP address for the VPN adapter and not your local network connection. The **VpnAdapter** property can be used to obtain the name of the active VPN adapter and the **VpnAddress** property can be used to determine the IP address assigned to the adapter.

Checking this property value may cause the current thread to block until the external IP address can be resolved and should never be used in conjunction with asynchronous socket connections.

Data Type

String

See Also

[HostAddress Property](#), [LocalAddress Property](#), [PeerAddress Property](#), [VpnAdapter Property](#), [VpnAddress Property](#)

Handle Property

Returns the descriptor for the current socket.

Syntax

object.**Handle**

Remarks

The **Handle** read-only property returns the descriptor of the socket being used by the control. If the control is not connected to a remote host, a value of -1 is returned. This property can be used in conjunction with direct calls to the Windows Sockets API.

When using Visual Studio .NET, you must use the property name **CtlHandle** instead.

Data Type

Integer (Int32)

See Also

[Connect Method](#), [Listen Method](#)

HashStrength Property

Return the length of the message digest that was selected.

Syntax

object.HashStrength

Remarks

The **HashStrength** property returns the number of bits used in the message digest (hash) that was selected. Common values returned by this property are 128 and 160. If this property returns a value of 0, this means that a secure connection has not been established with the server.

Data Type

Integer (Int32)

See Also

[CipherStrength Property](#), [Secure Property](#), [SecureCipher Property](#), [SecureHash Property](#), [SecureKeyExchange Property](#), [SecureProtocol Property](#)

HostAddress Property

Gets and sets the IP address of the remote host.

Syntax

object.HostAddress [= *ipaddress*]

Remarks

The **HostAddress** property can be used to set the IP address for a remote system that you wish to communicate with. If the address is valid and matches an entry in the host table, the **HostName** property will be changed to match the address.

Data Type

String

See Also

[AutoResolve Property](#), [HostFile Property](#), [HostName Property](#), [LocalAddress Property](#), [Resolve Method](#)

HostAlias Property

Returns the aliases defined for the current hostname.

Syntax

object.HostAlias(*Index*)

Remarks

The **HostAlias** property array returns the aliases assigned to the host specified by the **HostAddress** or **HostName** properties. If the host address or name can be resolved, the first element in the **HostAlias** array (an index value of 0) always refers to the host's fully qualified domain name. The end of the alias list is indicated when the property returns an empty string.

When using Visual Studio .NET, you must use the accessor method **get_HostAlias** instead of the property name, otherwise an error will be returned indicating that it not a member of the control class.

Data Type

String

Example

The following example places the all of the aliases for a specific host into a listbox:

```
Dim nIndex As Integer
```

```
List1.Clear
```

```
Socket1.HostName = Trim(Text1.Text)
```

```
Do While Len(Socket1.HostAlias(nIndex)) > 0
```

```
    List1.AddItem Socket1.HostAlias(nIndex)
```

```
    nIndex = nIndex + 1
```

```
Loop
```

See Also

[HostAddress Property](#), [HostName Property](#)

HostFile Property

Gets and sets the name of an alternate host file.

Syntax

object.HostFile [= *filename*]

Remarks

The **HostFile** property is used to specify the name of an alternate file for resolving hostnames and IP addresses. The host file is used as a database that maps an IP address to one or more hostnames, and is used when setting the **HostName** or **HostAddress** properties and establishing a connection with a remote host. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

ipaddress hostname [hostalias ...]

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

127.0.0.1 localhost

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Setting this property loads the file into memory allocated for the current thread. If the contents of the file have changed after the function has been called, those changes will not be reflected when resolving hostnames or addresses. To reload the host file from disk, set the property again with the same file name. To remove the alternate host file from memory, specify an empty string as the file name.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

Because the alternate host file is cached for the current thread, setting this property will affect all instances of the control in the same thread. For example, if a project has three instances of the control loaded on a form, setting the **HostFile** property will affect all three controls, not just the control that set the property. To determine if an alternate host file has been cached, check the property value. If the property returns an empty string, no alternate host file has been cached.

Data Type

String

See Also

[AutoResolve Property](#), [HostAddress Property](#), [HostName Property](#), [LocalName Property](#), [Resolve Method](#)

HostName Property

Gets and sets the name of the remote host.

Syntax

object.HostName [= *hostname*]

Remarks

The **HostName** property should be set to the name of the remote system that you wish to communicate with. If the name is found in the host table, the **HostAddress** property is updated to reflect the IP address of the host.

Note that it is legal to assign an IP address to this property, but it is not legal to assign a host name to the **HostAddress** property.

Data Type

String

See Also

[AutoResolve Property](#), [HostAddress Property](#), [HostFile Property](#), [LocalName Property](#), [Resolve Method](#)

InLine Property

Sets or returns if urgent data is received in-line with non-urgent data.

Syntax

object.**InLine** [= { True | False }]

Remarks

The **InLine** property controls how urgent (out-of-band) data is handled when reading data from the socket. If set to a value of true, urgent data is placed in the data stream along with non-urgent data. To determine if the data that is being read is urgent, the **AtMark** property can be read.

Urgent data is sent and received directly from the socket, and is not buffered even if buffering is enabled. It is recommended that you do not enable buffering if urgent data is being received in-line.

Data Type

Boolean

See Also

[NoDelay Property](#), [Urgent Property](#), [OnRead Event](#)

Interval Property

Gets and sets the number of milliseconds between calls to the control's **OnTimer** event.

Syntax

object.Interval [= *milliseconds*]

Remarks

The **Interval** property specifies the number of milliseconds between calls to the **OnTimer** event. A value of zero indicates that the timer is disabled and no events will be generated. The maximum interval value is 65536 milliseconds, which is slightly more than one minute.

Data Type

Integer (Int32)

See Also

[OnTimer Event](#)

IsBlocked Property

Return if the control is blocked performing an operation.

Syntax

object.IsBlocked

Remarks

The **IsBlocked** property returns True if the specified control is blocked performing an operation. Because the Windows Sockets API only permits one blocking operation per thread of execution, this property should be checked before starting any blocking operation.

If the **IsBlocked** property returns False, this means there are no blocking operations on the current thread at that time. If the property returns True, this tells you that you can't proceed with a socket operation. However, if the property returns False this does not guarantee that the next socket operation will not fail with a **swErrorOperationWouldBlock** or **swErrorOperationInProgress** error. The application should treat these errors as recoverable, and should be prepared to retry operations that result in them.

Note that this property will return True if there is *any* blocking operation being performed by the application, regardless of whether the control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

[Blocking Property](#), [LastError Property](#)

IsClosed Property

Determine if the connection has been closed by the remote host.

Syntax

object.IsClosed

Remarks

The **IsClosed** property returns True if the socket connection has been closed by the remote host. Note that it is possible to continue to receive data due to buffering.

Data Type

Boolean

See Also

[IsReadable Property](#), [IsWritable Property](#)

IsConnected Property

Determine if the control is connected to a remote host.

Syntax

object.IsConnected

Remarks

The **IsConnected** read-only property is set to a value of True if the control is connected with a remote host, otherwise the property has a value of false.

Data Type

Boolean

See Also

[Connect Method](#), [Disconnect Method](#), [OnConnect Event](#)

IsInitialized Property

Determine if the control has been initialized.

Syntax

object.IsInitialized

Remarks

The **IsInitialized** property is used to determine if the current instance of the control has been initialized properly. Normally this is done automatically when the control is loaded, however there are circumstances where the control may not be able to initialize itself. If this property returns **False**, the application must call the **Initialize** method to initialize the control before performing any other operation.

The most common reason that the control may not initialize correctly is that no valid development or runtime license key can be found or the license key that was provided is invalid. It may also indicate a problem with the system configuration or user access rights, such as not being able to load the required networking libraries or not being able to access the system registry.

Data Type

Boolean

See Also

[Initialize Method](#)

IsListening Property

Returns if the socket is listening for connections.

Syntax

object.IsListening

Remarks

The **IsListening** property returns True if the socket is listening for connections after the **Listen** method is called.

Data Type

Boolean

See Also

[Backlog Property](#), [Listen Method](#), [OnAccept Event](#)

IsReadable Property

Determine if data can be read from the socket without blocking.

Syntax

object.IsReadable

Remarks

The **IsReadable** property returns True if data can be read from the socket without blocking. For non-blocking sockets, this property can be checked before the application attempts to read the socket, preventing an error.

Data Type

Boolean

See Also

[IsClosed Property](#), [IsWritable Property](#), [Peek Method](#), [Read Method](#), [OnRead Event](#)

IsWritable Property

Determine if data can be written to the socket without blocking.

Syntax

object.IsWritable

Remarks

The **IsWritable** property returns True if data can be written to the socket without blocking. For non-blocking sockets, this property can be checked before the application attempts to write to the socket, preventing an error.

If the **IsWritable** property returns False, this means that the application cannot write to the socket at that time. However, if the property returns True, this does not guarantee that you will be able to write to the socket without an error. The next socket operation may result in a **swErrorOperationWouldBlock** or **swErrorOperationInProgress** error. The application should treat these errors as recoverable, and should be prepared to retry operations that result in them.

Data Type

Boolean

See Also

[IsClosed Property](#), [IsReadable Property](#), [OnWrite Event](#)

KeepAlive Property

Set or return if keep-alive packets are sent on a connected socket.

Syntax

object.KeepAlive [= { True | False }]

Remarks

Setting the **KeepAlive** property to a value of true indicates that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This property can only be set for stream sockets that were created with the **Protocol** property set to a value of **swProtocolTcp**.

If this property is set to true, keep-alive packets will start being generated five seconds after the socket has become idle with no data being sent or received. Enabling this option can be used by applications to detect when a physical network connection has been lost. However, it is recommended that most applications query the remote host directly to determine if the connection is still active. This is typically accomplished by sending specific commands to the server to query its status, or checking the elapsed time since the last response from the server.

Data Type

Boolean

See Also

[Broadcast Property](#), [InLine Property](#), [NoDelay Property](#), [Protocol Property](#), [ReuseAddress Property](#), [Route Property](#)

LastError Property

Gets and sets the last error that occurred on the control.

Syntax

object.**LastError** [= *errorcode*]

Remarks

The **LastError** property can be read to determine the last error that occurred for this control. If a value is assigned to this property, it must either be zero (to clear the error) or a valid error code for the control.

Data Type

Integer (Int32)

See Also

[LastErrorString Property](#), [ThrowError Property](#), [OnError Event](#)

LastErrorString Property

Return a description of the last error that occurred.

Syntax

object.LastErrorString

Remarks

The **LastErrorString** property returns a string that contains a description of the last error that occurred.

Data Type

String

See Also

[LastError Property](#), [ThrowError Property](#), [OnError Event](#)

Linger Property

Gets and sets the number of seconds to wait for the socket to close.

Syntax

object.Linger [= *seconds*]

Remarks

Setting the **Linger** property to a value greater than zero indicates that the **Disconnect** method should wait up to the specified number of seconds for any data on the socket to be written before it is closed. A value of zero indicates that the socket should be closed immediately (but gracefully, without data loss).

Data Type

Integer (Int32)

See Also

[OnDisconnect Event](#)

LocalAddress Property

Return the IP address of the local host.

Syntax

object.LocalAddress

Remarks

The **LocalAddress** read-only property returns the local host's IP address in dot notation, as four numbers separated by periods.

Data Type

String

See Also

[AutoResolve Property](#), [ExternalAddress Property](#), [HostAddress Property](#), [LocalName Property](#), [Bind Method](#)

LocalName Property

Return the name of the local host.

Syntax

object.LocalName

Remarks

The **LocalName** read-only property returns the fully qualified domain name of the local system. This consists of the local computer name and its domain name. The actual value returned depends on the system configuration. If no domain has been specified for the system, then only the machine name will be returned.

Data Type

String

See Also

[AutoResolve Property](#), [HostName Property](#), [LocalAddress Property](#), [Resolve Method](#)

LocalPort Property

Gets and sets the port number for a local listening socket.

Syntax

object.LocalPort [= *port*]

Remarks

The **LocalPort** property is used to set the port number that a server will listen on for connections.

Data Type

Integer (Int32)

See Also

[PeerPort Property](#), [RemotePort Property](#), [ReservedPort Property](#), [Bind Method](#), [Listen Method](#)

NoDelay Property

Enable or disable the Nagle algorithm.

Syntax

object.NoDelay [= { True | **False** }]

Remarks

The **NoDelay** property is used to enable or disable the Nagle algorithm, which buffers unacknowledged data and ensures that a full-size packet can be sent to the remote host. By default this property value is set to False, which enables the Nagle algorithm (in other words, the data being written may not actually be sent until it is optimal to do so). Setting this property to True disables the Nagle algorithm, minimizing the time delays between the data packets being sent.

This property should be set to True only if it is absolutely required and the implications of doing so are understood. Disabling the Nagle algorithm can have a significant negative impact on the performance of the application.

Data Type

Boolean

See Also

[InLine Property](#), [KeepAlive Property](#), [ReuseAddress Property](#), [Route Property](#)

PeerAddress Property

Return the IP address of the remote peer.

Syntax

object.PeerAddress

Remarks

The **PeerAddress** property returns the IP address of the remote system that the local host is connected to. If a datagram socket is being used, this property will return the address of the system which sent the last datagram that was read. If no connection has been established, this property will return 255.255.255.255.

If this property is read inside an **OnAccept** event handler, it will return the IP address of the client that is requesting the connection. The application may use this information to determine if it wishes to accept or reject the client connection. If the IP address information is not available for the client at that time, this property will return the address 0.0.0.0.

Data Type

String

See Also

[HostAddress Property](#), [LocalAddress Property](#), [PeerName Property](#), [PeerPort Property](#)

PeerName Property

Return the name of the remote peer.

Syntax

object.PeerName

Remarks

The **PeerName** property returns the name of the remote system that the local host is connected to. If a datagram socket is being used, this property will return the name of the system which sent the last datagram that was read.

Accessing this property causes the control to perform a blocking reverse DNS lookup, attempting to match the client Internet address with a hostname. Not all addresses have a reverse DNS record, in which case this property will return an empty string. It is recommended that most applications use the value of the **PeerAddress** property rather than use the **PeerName** property to distinguish between connections from a remote host.

Data Type

String

See Also

[HostName Property](#), [LocalName Property](#), [PeerAddress Property](#), [PeerPort Property](#)

PeerPort Property

Return the port number of the remote connection or datagram.

Syntax

object.**PeerPort**

Remarks

The **PeerPort** property returns the port number that the remote host has used when establishing a connection with the local system. If a datagram socket is being used, this property will return the port number used by the remote host which sent the last datagram that was received.

Data Type

String

See Also

[LocalPort Property](#), [PeerAddress Property](#), [PeerName Property](#), [RemotePort Property](#)

PhysicalAddress Property

Return the MAC address for the local host's Ethernet or Token Ring adapter.

Syntax

object.PhysicalAddress

Remarks

The **PhysicalAddress** property returns the Media Access Control (MAC) address for an Ethernet or Token Ring network adapter installed and configured on the local system. Since it is guaranteed that every adapter is assigned a unique address throughout the world, this value can be safely used for identification purposes. It is possible that this property will return an empty string, which indicates that it could not find a network adapter.

If more than one physical network adapter is installed on the system, this property will return the MAC address of the first adapter that it finds.

Data Type

String

See Also

[AdapterAddress Property](#), [AdapterCount Property](#), [LocalAddress Property](#)

Protocol Property

Gets and sets the protocol that should be used to create the socket.

Syntax

object.Protocol [= *protocol*]

Remarks

The **Protocol** property specifies the type of socket that is to be created. This property may only be set before a socket has been created, or after it has been closed. Supported socket protocols are:

Value	Description
swProtocolTcp	Specifies the User Datagram Protocol. This is a stateless, peer-to-peer message oriented protocol, with the data sent in discrete packets. UDP is a simpler network protocol that does not have the inherent reliability of TCP, but it has less overhead and is ideal for real-time applications where a dropped packet is preferable to the delay of waiting for a packet to be retransmitted. It also supports the broadcasting of datagrams over local networks, and is commonly used by services that must exchange a relatively small amount of data with a large number of clients.
swProtocolUdp	Specifies the User Datagram Protocol. This is a stateless, peer-to-peer message oriented protocol, with the data sent in discrete packets. UDP is a simpler network protocol that does not have the inherent reliability of TCP, but it has less overhead and is ideal for real-time applications where a dropped packet is preferable to the delay of waiting for a packet to be retransmitted. It also supports the broadcasting of datagrams over local networks, and is commonly used by services that must exchange a relatively small amount of data with a large number of clients.
swProtocolRaw	Raw sockets. This socket type is for special purpose applications which need access to the IP datagram. It is not supported on all platforms and should only be used if required.

The default value for this property is **swProtocolTcp**.

Data Type

Integer (Int32)

See Also

[Bind Method](#), [Connect Method](#)

RemotePort Property

Gets and sets the port number for a remote connection.

Syntax

object.RemotePort [= *portno*%]

Remarks

The **RemotePort** property is used to set the port number that the control will use to establish a connection with the remote host.

Data Type

Integer (Int32)

See Also

[HostAddress Property](#), [HostName Property](#), [LocalPort Property](#)

ReservedPort Property

Set or return if a reserved local port number should be allocated.

Syntax

object.ReservedPort [= { True | False }]

Remarks

The **ReservedPort** property determines if a reserved local port number is used by the control when the socket is created (reserved port numbers are in the range of 513 through 1023, inclusive). Some application protocols require that the client bind to a local port number in this range. By setting the **LocalPort** property to 0 and the **ReservedPort** property to True, a reserved port number will be used when the socket is created. The default value for this property is False, which specifies that a standard port number with a value of 1024 or higher will be bound to the socket unless the **LocalPort** property is explicitly set to a non-zero value. Reserved ports should only be used by those applications that expressly need them to implement a specific protocol.

It is possible that the error **swErrorAddressInUse** will be returned when attempting to connect using a reserved port number. The value of the **LocalPort** property will contain the reserved port number that could not be used.

Data Type

Boolean

See Also

[LocalPort Property](#), [RemotePort Property](#)

ReuseAddress Property

Set or return if a local socket address can be reused by the application.

Syntax

`object.ReuseAddress [= { True | False } ]`

Remarks

Setting this property to a value of true allows the address that the socket is listening on to be reused.

When a listening socket is closed, the socket will normally go into a TIME-WAIT state where the local address and port number cannot be immediately reused. A consequence of this is that calling the **Disconnect** method immediately followed by the **Listen** method using the same address and port number values may result in an error indicating that the specified address is already in use. By setting this property to True, that error is avoided and the listening socket can be created immediately without waiting for the TIME-WAIT period to elapse.

Data Type

Boolean

See Also

[Broadcast Property](#), [InLine Property](#), [NoDelay Property](#), [KeepAlive Property](#), [Route Property](#)

Secure Property

Set or return if a connection to the server is secure.

Syntax

object.Secure [= { True | False }]

Remarks

The **Secure** property determines if a secure connection is established to the server. The default value for this property is False, which specifies that a standard connection to the server is used. To establish a secure connection, the application should set this property value to True prior to calling the **Connect** method. Once the connection has been established, the client may request files or submit queries to the server as with standard connections.

It is possible for an application to establish a non-secure connection, and then switch to a secure connection at some later point during the session. Initially set the **Secure** property to False, then connect to the server normally. Once the connection has been established, setting the **Secure** property to True will cause the control to negotiate a secure connection with the remote host. If the socket was created using the **Accept** method, the control will block and wait for the client to begin the negotiation. If the socket was created using the **Connect** method, it will immediately begin the negotiation with the server. Note that if a non-blocking (asynchronous) socket is being used, the application must wait to set the **Secure** property to True after the **OnConnect** event has fired.

Setting the **Secure** property to False during a connection will cause the control to send a shutdown message to the remote host. This may cause which may cause it to terminate the connection, however it will not close the socket. It is recommended that applications do not set the **Secure** property to False after a secure connection has been established, and instead use the **Disconnect** method to close the connection.

It is recommended that the application use exception handling to catch any errors that may occur when changing the value of this property. If the control is unable to initialize the Windows security libraries, an exception will be thrown when this property value is modified.

Data Type

Boolean

Example

The following example establishes a secure connection to a web server:

```
SocketWrench1.HostName = strHostName
SocketWrench1.RemotePort = 443
SocketWrench1.Secure = True

nError = SocketWrench1.Connect()
If nError > 0 Then
    MsgBox "Unable to connect to server " & strHostName, vbExclamation
    Exit Sub
End If

If SocketWrench1.CertificateStatus <> swCertificateValid Then
    nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
        "Are you sure you wish to continue?", vbYesNo)

    If nResult = vbNo Then
```

```
        SocketWrench1.Disconnect
    Exit Sub
End If
End If
```

See Also

[CertificateExpires Property](#), [CertificateIssued Property](#), [CertificateIssuer Property](#), [CertificateStatus Property](#), [CertificateSubject Property](#), [CipherStrength Property](#), [HashStrength Property](#), [SecureCipher Property](#), [SecureHash Property](#), [SecureKeyExchange Property](#), [SecureProtocol Property](#), [Connect Method](#)

SecureCipher Property

Return the encryption algorithm used to establish the secure connection with the server.

Syntax

`object.SecureCipher`

Remarks

The **SecureCipher** property returns an integer value which identifies the algorithm used to encrypt the data stream. This property may return one of the following values:

Value	Description
swCipherNone	No cipher has been selected. This is not a secure connection with the server.
swCipherRC2	The RC2 block cipher was selected. This is a variable key length cipher which supports keys between 40- and 128-bits in length, in 8-bit increments.
swCipherRC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40- and 128-bits in length, in 8-bit increments.
swCipherRC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
swCipherDES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher using 56-bit keys.
swCipherDES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively using a 168-bit key length.
swCipherDESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
swCipherAES	The Advanced Encryption Standard cipher (also known as the Rijndael cipher) is a fixed block size cipher which use a key size of 128, 192 or 256 bits. This cipher is supported on Windows XP SP3 and later versions of the operating system.
swCipherSkipjack	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
swCipherBlowfish	The Blowfish block cipher was selected. This is a variable key length cipher up to 448 bits, using a 64-bit block size.

If a secure connection has not been established, this property will return a value of zero.

Data Type

Integer (Int32)

See Also

[CipherStrength Property](#), [HashStrength Property](#), [Secure Property](#), [SecureHash Property](#), [SecureKeyExchange Property](#), [SecureProtocol Property](#)

SecureHash Property

Return the message digest selected when establishing the secure connection with the server.

Syntax

object.SecureHash

Remarks

The **SecureHash** property returns an integer value which identifies the message digest algorithm that was selected when a secure connection is established. This property may return one of the following values:

Value	Description
swHashMD5	The MD5 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
swHashSHA1	The SHA-1 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
swHashSHA256	The SHA-256 algorithm has been selected.
swHashSHA384	The SHA-384 algorithm has been selected.
swHashSHA512	The SHA-512 algorithm has been selected.

If a secure connection has not been established, this property will return a value of zero.

Data Type

Integer (Int32)

See Also

[CipherStrength Property](#), [HashStrength Property](#), [Secure Property](#), [SecureCipher Property](#), [SecureKeyExchange Property](#), [SecureProtocol Property](#)

SecureKeyExchange Property

Return the key exchange algorithm used to establish the secure connection with the server.

Syntax

`object.SecureKeyExchange`

Remarks

The **SecureKeyExchange** property returns an integer value which identifies the key-exchange algorithm used when establishing a secure connection. This property may return one of the following values:

Value	Description
swKeyExchangeNone	No key exchange algorithm has been selected. This is not a secure connection with the server.
swKeyExchangeRSA	The RSA public key exchange algorithm has been selected.
swKeyExchangeKEA	The KEA public key exchange algorithm has been selected. This is an improved version of the Diffie-Hellman public key algorithm.
swKeyExchangeDH	The Diffie-Hellman public key exchange algorithm has been selected.
swKeyExchangeECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography. This key exchange algorithm is only supported on Windows XP SP3 and later versions of the operating system.

If a secure connection has not been established, this property will return a value of zero.

Data Type

Integer (Int32)

See Also

[CipherStrength Property](#), [HashStrength Property](#), [Secure Property](#), [SecureCipher Property](#), [SecureHash Property](#), [SecureProtocol Property](#)

SecureProtocol Property

Gets and sets the security protocol used to establish the secure connection with the server.

Syntax

object.SecureProtocol [= *protocol*]

Remarks

The **SecureProtocol** property can be used to specify the security protocol to be used when establishing a secure connection with a server. By default, the control will attempt to use TLS 1.3 to establish the connection. If TLS 1.3 is not supported, TLS 1.2 will be used. The appropriate protocol is automatically selected based on the capabilities of both the client and server.

It is recommended that you only change this property value if you fully understand the implications of doing so. Assigning a value to this property will override the default and force the control to attempt to use only the protocol specified. One or more of the following values may be used:

Value	Description
stProtocolNone	No security protocol has been selected. A secure connection has not been established.
stProtocolTLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
stProtocolTLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
stProtocolTLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
stProtocolTLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

Multiple security protocols may be specified by combining them using a bitwise Or operator. After a connection has been established, reading this property will identify the protocol that was selected to establish the connection. Attempting to set this property after a connection has been established will result in an exception being thrown. This property should only be set after setting the **Secure** property to True and before calling the **Connect** method.

Data Type

Integer (Int32)

See Also

[CipherStrength Property](#), [HashStrength Property](#), [Secure Property](#), [SecureCipher Property](#), [SecureHash Property](#), [SecureKeyExchange Property](#)

ThrowError Property

Enables or disables error handling by the application.

Syntax

object.ThrowError = { True | False }

Remarks

Error handling for methods can be performed using either return codes or exceptions, depending on the value of this property.

If the **ThrowError** property is set to False, methods will return a status value indicating success or failure. Applications should check the return value of each method and use the **LastError** and **LastErrorString** properties to determine the cause of any error.

If the **ThrowError** property is set to True, errors occurring within the control will be raised to the host application as ActiveX exceptions. In addition, the **OnError** event will fire. This allows applications to use language-specific exception handling mechanisms, such as Visual Basic error handling, instead of checking individual return values after each method call.

Even when the **ThrowError** property is set to True, the **LastError** and **LastErrorString** properties will still be updated with information about the most recent error.

If an error occurs while accessing a property value, an exception will be raised regardless of the value of the **ThrowError** property. However, the **OnError** event will not be fired for property access errors.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
Dim lResult As Long

Socket1.ThrowError = False
lResult = Socket1.Connect

If lResult <> 0 Then
    MsgBox "Error on Connect: " & lResult
    Exit Sub
End If
```

The following example handles errors by throwing them to the container (VB):

```
On Error Resume Next: Err.Clear

Socket1.ThrowError = True
Socket1.Connect

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
End If
On Error GoTo 0
```

See Also

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.Timeout [= *seconds*]

Remarks

Setting this property specifies the number of seconds until a blocking operation fails and the control returns an error.

For backwards compatibility with previous versions of the control, if a value greater than 1000 is specified when setting the property, the control assumes that milliseconds were intended and adjusts the value accordingly.

Data Type

Integer (Int32)

See Also

[LastError Property](#), [OnError Event](#), [OnTimeout Event](#)

Trace Property

Enable or disable network function level logging.

Syntax

`object.Trace [= { True | False } ]`

Remarks

The **Trace** property is used to enable or disable the logging of network function calls. When enabled, each function call is logged to a file, including the function parameters, return value and error code if applicable. This facility can be enabled and disabled at run time, and the trace log file can be specified by setting the **TraceFile** property. All function calls that are being logged are appended to the trace file, if it exists. If no log file exists when tracing is enabled, the file is created.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

Data Type

Boolean

See Also

[TraceFile Property](#), [TraceFlags Property](#)

TraceFile Property

Gets and sets the name of the file to use when logging network function calls.

Syntax

`object.TraceFile [= filename ]`

Remarks

The **TraceFile** property is used to specify the name of the file that is created when logging is enabled. If this property is set to an empty string (the default value), then a file named **SocketTools.log** is created in the current user's Documents folder. It is recommended that you always specify a complete path to the log file. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

If the file exists, the output is appended to the file, otherwise the file is created. Since logging is enabled per-process, the log file is shared by all instances of the controls being used. If you create multiple instances of the control, you only need to enable logging for a single instance. Because log files can grow very quickly, even with modest applications, it is recommended that you delete the file when it is no longer needed.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "0x". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxx:xxxx:xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, the control is attempting to connect to a system with the IPv4 address 192.0.0.1 on port 1234.

If the log file cannot be created, or the user does not have permission to modify an existing file, the error is silently ignored and no output will be generated.

Data Type

String

See Also

TraceFlags Property

Gets and sets the network function logging flags.

Syntax

object.TraceFlags [= *flags*]

Remarks

The **TraceFlags** property is used to specify the type of information written to the trace file when socket function tracing is enabled. The following values may be used:

Value	Description
dnsTraceInfo	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
dnsTraceError	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
dnsTraceWarning	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
dnsTraceHexDump	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
dnsTraceTruncate	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.

Since logging is enabled per-process, the trace flags are shared by all instances of the control being used. If multiple controls have tracing enabled, the **TraceFlags** property should be set to the same value for each control. Changing the trace flags for any one instance of the control will affect the logging performed for all other instances used by the application.

Warnings are generated when a non-fatal error is returned by a Windows Sockets function. For example, if data is being written through the control and an error indicating that the operation would block is returned, only a warning is logged since the application simply needs to attempt to write the data at a later time.

Data Type

Integer (Int32)

See Also

[Trace Property](#), [TraceFile Property](#)

Urgent Property

Send or receive urgent data.

Syntax

object.**Urgent** [= { True | False }]

Remarks

This boolean property affects how the **Read** and **Write** methods read or write data to the socket. If set to a value of true, urgent (out-of-band) data will be read or written. All reads or writes of urgent data are unbuffered. The property value will automatically be reset to a value of false after the socket has been read or written.

Note: Not all implementations may support more than one byte of urgent data if the data is not being received in-line. Refer to the **InLine** property for additional information.

Data Type

Boolean

See Also

[InLine Property](#), [OnRead Event](#)

Version Property

Return the current version of the control.

Syntax

object.Version

Remarks

The **Version** property returns the current version of the object. This can be used by an application for validation purposes. The version returned is composed of four numbers, separated by periods. The first number is the major version number, the second number is the minor version number, the third is the build number and the fourth is the revision number.

Data Type

String

VpnAdapter Property

Return the name of the active VPN adapter.

Syntax

object.VpnAdapter

Remarks

The **VpnAdapter** property attempts to detect if the system is currently using a Virtual Private Network (VPN) connection. The detection logic is based on identifying active network adapters with names or descriptions that match known VPN software or common VPN adapter naming conventions, while explicitly excluding adapters associated with virtualization platforms (such as Hyper-V, VMware, and VirtualBox), loopback interfaces, and other non-VPN virtual adapters.

Because there is no industry standard for how VPN clients are identified at the network adapter level, this function uses a heuristic approach and cannot guarantee detection in every scenario. Some VPN clients may use custom adapter names or attempt to obscure their presence, and new VPN software may not be recognized until the internal detection list is updated. There is also a small chance that non-VPN software using similar naming conventions could result in a false positive.

The value of this property should be treated as a hint for diagnostic or troubleshooting purposes. It should not be used to restrict access to your application or deny service to users based solely on the possible presence of a VPN connection. If you wish to determine the IP address assigned to the VPN adapter, check the value of the **VpnAddress** property.

If no active VPN adapter is detected, this property will return an empty string.

Data Type

String

See Also

[ExternalAddress Property](#), [HostAddress Property](#), [LocalAddress Property](#), [VpnAddress Property](#)

VpnAddress Property

Return the local IP address assigned to the active VPN adapter.

Syntax

object.VpnAddress

Remarks

The **VpnAddress** property returns the IP address assigned to the active VPN connection. This property uses heuristics to detect if the system is currently using a Virtual Private Network (VPN) connection. Because there is no industry standard for how VPN adapters are identified, it is possible a VPN may not be detected and the property will return an empty string. If the property does return an IP address, the **VpnAdapter** property will return the name of the VPN adapter.

If this property returns an IP address, it will be the local address bound to the VPN adapter, not the external IP address. You can check the value of the **ExternalAddress** property to determine what external IP address has been assigned to the VPN adapter. By default, this property will prefer IPv4 addresses and will only return an IPv6 address if the VPN adapter has not been assigned an IPv4 address.

Data Type

String

See Also

[ExternalAddress Property](#), [HostAddress Property](#), [LocalAddress Property](#), [VpnAdapter Property](#)

SocketWrench Control Methods

Method	Description
Abort	Terminate the connection with a remote host
Accept	Accepts a client connection on a listening socket
Bind	Bind the socket to the specified local address and port number
Cancel	Cancels the current blocking network operation
Connect	Establish a connection with a server
ConnectUrl	Establish a connection with a server using a URL
Disconnect	Terminate the connection with a remote host
Flush	Flush the contents of the send and receive socket buffers
Initialize	Initialize the control and validate the runtime license key
Listen	Listen for incoming connections
Peek	Return data read from the socket, but do not remove it from the socket buffer
Read	Return data read from the socket
ReadByte	Read a single byte of data from the socket
ReadLine	Read a line of data from the socket, storing it in a string buffer
ReadStream	Read a stream of data from the socket, returning when all data has been read
Reject	Reject a pending client connection
Reset	Reset the internal state of the control
Resolve	Resolves a host name to a host IP address
Shutdown	Stop sending or receiving data on the socket
StoreStream	Read a stream of data from the remote host, storing it in a file
Uninitialize	Uninitialize the control and release any system resources that were allocated
Write	Write data to the socket
WriteByte	Write a single byte of data to the socket
WriteLine	Write a line of data to the socket, terminated with a carriage-return and linefeed
WriteStream	Write a stream of data to the socket, returning when all data has been written

Abort Method

Terminate the connection with a remote host.

Syntax

object.Abort

Parameters

None.

Return Value

A value of zero is returned if the connection was terminated successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Abort** method immediately closes the socket, without waiting for any remaining data to be written out. This method should only be used when the connection must be closed immediately before the application terminates.

See Also

[Connect Method](#), [Disconnect Method](#)

Accept Method

Accepts a client connection on a listening socket.

Syntax

object.Accept (*Handle*, [*Options*])

Parameters

Handle

An integer value that specifies the handle of the listening socket. If the control that invokes this method is not the listening socket, then the listening socket may continue to listen for incoming connections. If the control invokes this method using its own **Handle** property, it will stop listening for connections.

Options

An optional integer value that specifies one or more options. If this parameter is omitted, the values of the properties listed below will be used to determine the default options when accepting the connection.

Value	Property
swOptionNone	None
swOptionDontRoute	Route = False
swOptionKeepAlive	KeepAlive = True
swOptionReuseAddress	ReuseAddress = True
swOptionNoDelay	NoDelay = True
swOptionInLine	InLine = True
swOptionSecure	Secure = True

Return Value

A value of zero is returned if the acceptance was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

To set the **Options** argument explicitly, set it as a combination of values chosen from the table above. Use the appropriate constant if you wish the attribute corresponding to the property to be True, except for the **Route** property. Specifying the *swOptionDontRoute* option is the same as setting the Route property to a value of False.

See Also

[Backlog Property](#), [Handle Property](#), [Listen Method](#), [Reject Method](#), [OnAccept Event](#)

Bind Method

Bind the socket to the specified local address and port number.

Syntax

object.Bind([*LocalAddress*], [*LocalPort*], [*Protocol*], [*Timeout*], [*Options*])

Parameters

LocalAddress

An optional string value that specifies the local Internet address that the socket should be bound to. To bind to any valid network interface on the local system, specify the address 0.0.0.0. Applications should only specify a particular address if it is absolutely necessary. In most cases a local address is not required when establishing a client connection. If this value is not specified, the **LocalAddress** property will be used to determine the default value.

LocalPort

An optional integer value that specifies a local port number that the socket should be bound to. To bind to any available port number, specify a port number of 0. Applications should only specify a particular port number if it is absolutely necessary. The maximum valid port number is 65535. If this argument is not specified, the **LocalPort** property will be used to determine the default value.

Protocol

An optional integer value that specifies the protocol that should be used when establishing the connection. If this argument is not specified, the value of the **Protocol** property will be used as the default. One of the following values may be used:

Value	Description
swProtocolTcp	Specifies the Transmission Control Protocol. This protocol provides a reliable means of communication between two computers using a client/server architecture. The data is exchanged as a stream of bytes, with the protocol ensuring that the data arrives in the same byte order that it was sent, without duplication or missing data. This protocol is designed for accuracy and not speed, therefore TCP can sometimes incur relatively long delays while waiting for out-of-order packets and the retransmission of data which can make it unsuitable for some applications such as streaming video or audio.
swProtocolUdp	Specifies the User Datagram Protocol. This is a stateless, peer-to-peer message oriented protocol, with the data sent in discrete packets. UDP is a simpler network protocol that does not have the inherent reliability of TCP, but it has less overhead and is ideal for real-time applications where a dropped packet is preferable to the delay of waiting for a packet to be retransmitted. It also supports the broadcasting of datagrams over local networks, and is commonly used by services that must exchange a relatively small amount of data with a large number of clients.

Timeout

An optional integer value that specifies the amount of time until a blocking operation fails. If this argument is not specified, the **Timeout** property will be used to determine the default value.

Options

An optional integer value that specifies one or more options which are to be used when establishing the connection. The value is created by combining the options using a bitwise Or operator. Note that if this argument is specified, it will override any property values that are related to that option.

Value	Description
swOptionBroadcast	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
swOptionDontRoute	This option specifies default routing should not be used. This option should not be specified unless absolutely necessary.
swOptionKeepAlive	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This option is only valid for stream sockets.
swOptionReuseAddress	This option specifies the local address can be reused. This option is commonly used by server applications.
swOptionNoDelay	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.
swOptionInLine	This option specifies that out-of-band data should be received inline with the standard data stream. This option is only valid for stream sockets.

Return Value

A value of zero is returned if the connection was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

When this method is called with **swProtocolUdp** as the specified protocol, it will immediately create the datagram socket and bind it to the given address. When this method is called with **swProtocolTcp** as the specified protocol, creation of the socket is deferred until the **Connect** method is called. For stream sockets, this method will set the local address, port number and default options used when the socket is actually created.

See Also

[Broadcast Property](#), [LocalAddress Property](#), [LocalPort Property](#), [Timeout Property](#), [Connect Method](#), [Disconnect Method](#)

Cancel Method

Cancels the current blocking network operation.

Syntax

object.Cancel

Parameters

None.

Return Value

None.

Remarks

The **Cancel** method cancels any blocking network operation in the current thread. This is typically used inside an event handler, causing the blocking method to return to the caller with an error indicating that the current operation was canceled. This method sets an internal flag that is periodically checked during a blocking operation, such as waiting for more data to arrive. If the current thread is not blocked at the time that this method is called, it will have no effect.

See Also

[Reset Method](#)

Connect Method

Establish a connection with a server.

Syntax

```
object.Connect( [RemoteAddress], [RemotePort], [Protocol], [Timeout], [Options], [LocalAddress],  
[LocalPort] )
```

Parameters

RemoteAddress

An optional string value that specifies the host name or IP address of the server. If this parameter is omitted, it defaults to the value of the **HostAddress** property if it is defined; otherwise, it defaults to the value of the **HostName** property.

RemotePort

An optional integer value that specifies the port number that the server is using to listen for connections. If this parameter is omitted, the **RemotePort** property will be used to determine the default value.

Protocol

An optional integer value that specifies the protocol that should be used when establishing the connection. If this parameter is omitted, the **Protocol** property will be used to determine the default value. It may be one of the following values:

Value	Description
swProtocolTcp	The connection will use the Transmission Control Protocol. This protocol provides a reliable means of communication between two computers using a client/server architecture. The data is exchanged as a stream of bytes, with the protocol ensuring that the data arrives in the same byte order that it was sent, without duplication or missing data. This protocol is designed for accuracy and not speed, therefore TCP can sometimes incur relatively long delays while waiting for out-of-order packets and the retransmission of data which can make it unsuitable for some applications such as streaming video or audio.
swProtocolUdp	The connection will use the User Datagram Protocol. This is a stateless, peer-to-peer message oriented protocol, with the data sent in discrete packets. UDP is a simpler network protocol that does not have the inherent reliability of TCP, but it has less overhead and is ideal for real-time applications where a dropped packet is preferable to the delay of waiting for a packet to be acknowledged or retransmitted. It also supports the broadcasting of datagrams over local networks, and is commonly used by services that must exchange a relatively small amount of data with a large number of clients.

Timeout

An optional integer value that specifies the amount of time until a blocking operation fails. If this parameter is omitted, the **Timeout** property will be used to determine the default value.

Options

An optional integer value that specifies one or more socket options which are to be used when

establishing the connection. The value is created by combining the options using a bitwise Or operator. Note that if this argument is specified, it will override any property values that are related to that option.

Value	Description
swOptionBroadcast	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
swOptionDontRoute	This option specifies default routing should not be used. This option should not be specified unless absolutely necessary.
swOptionKeepAlive	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
swOptionNoDelay	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
swOptionInLine	This option specifies that out-of-band data should be received inline with the standard data stream. This option is only valid for stream sockets.
swOptionTrustedSite	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the TLS protocol.
swOptionSecure	This option specifies that a secure connection should be established with the remote host. The specific version of TLS can be specified by setting the SecureProtocol property. By default, the connection will use TLS 1.2 and the strongest cipher suites available. Older versions of Windows prior to Windows 7 and Windows Server 2008 R2 only support TLS 1.0 and secure connections will automatically downgrade on those platforms.
swOptionSecureFallback	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
swOptionPreferIPv6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.

LocalAddress

An optional string value that specifies the local IP address of the network adapter that the control

should use when establishing the connection. If this parameter is omitted, the control will bind to any suitable adapter on the local system. It is recommended that you omit this parameter when establishing a TCP connection unless you fully understand the implications of binding the socket to a specific local address.

LocalPort

An optional integer value that specifies the local port number that the control should use when establishing the connection. If this argument is not specified, an appropriate local port number will be automatically allocated for the connection. It is recommended that you omit this parameter when establishing a TCP connection unless you fully understand the implications of binding the socket to a specific local port.

Return Value

A value of zero is returned if the connection was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

See Also

[HostName Property](#), [KeepAlive Property](#), [NoDelay Property](#), [Options Property](#), [RemotePort Property](#), [ReuseAddress Property](#), [Route Property](#), [Secure Property](#), [SecureProtocol Property](#), [Timeout Property](#), [Bind Method](#), [Disconnect Method](#), [OnConnect Event](#), [OnDisconnect Event](#)

ConnectUrl Method

Establish a connection with a server using a URL.

Syntax

object.ConnectUrl([*Url*], [*Timeout*], [*Options*])

Parameters

Url

A string value which specifies a URL used when establishing the connection. This parameter cannot be omitted and it cannot be an empty string. If a non-standard URI scheme is used, the port number must be explicitly specified or the method will fail. See the remarks below for more information on the format supported by this method.

Timeout

An optional integer value that specifies the amount of time until a blocking operation fails. If this parameter is omitted, the **Timeout** property will be used to determine the default value.

Options

An optional integer value that specifies one or more socket options which are to be used when establishing the connection. The value is created by combining the options using a bitwise Or operator. Note that if this argument is specified, it will override any property values that are related to that option.

Value	Description
swOptionKeepAlive	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
swOptionNoDelay	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
swOptionInLine	This option specifies that out-of-band data should be received inline with the standard data stream. This option is only valid for stream sockets.
swOptionTrustedSite	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the TLS protocol.
swOptionSecure	This option specifies that a secure connection should be established with the remote host. The specific version of TLS can be specified by setting the SecureProtocol property. By default, the connection will use TLS 1.2 and the strongest cipher suites available.
swOptionSecureFallback	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this

	option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
swOptionPreferIPv6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.

Return Value

A value of zero is returned if the connection was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ConnectUrl** method provides a simplified interface which can be used to establish a connection using a URL. This method can only be used to establish connections using TCP and does not currently support the use of URLs to connect with a service which uses UDP. The general format of the URL should look like this:

```
[scheme]:// [[username : password] @] hostname [:port] / [path;parameters ...]
```

This method recognizes most standard URI schemes which use this format. The host name and port number specified in the URL will be used to establish a connection and the remaining information will be discarded. If the URL does not explicitly specify a port number, the default port number associated with the scheme will be used as the default value. For example, consider the following:

```
https://www.example.com
```

In this example, there is no port number specified; instead, the default port for the **https://** scheme would be used, which is port 443. The host name **www.example.com** would be resolved into an IP address and the connection established on port 443. This method will also recognize a simpler format which only includes the host name and port number without a URI scheme, such as:

```
www.example.com:443
```

When used in this way, the port number must always be provided. Without a URI scheme or an explicit port number, the method cannot determine what port number should be used when establishing the connection. The same also applies if a custom, non-standard URI scheme is provided which is not recognized.

If the URI scheme specifies a secure protocol which requires implicit TLS, this method will automatically enable security options. For example, providing a URL which uses the **https://** scheme will automatically enable a secure connection regardless if the **Options** parameter includes that option. If a URI scheme is used in conjunction with a port number associated with a secure service, security will also be enabled for that connection. For example:

```
http://www.example.com:443
```

The standard **http://** scheme is used which does not indicate a secure connection. However, since port 443 is the standard port designated for a secure HTTP connection, a secure connection will be enabled by default, even if **swOptionSecure** has not been specified by the caller. Alternatively, if a custom port number is specified in the URL or the scheme is not recognized as one which requires implicit TLS, security options will not be automatically enabled for the connection.

The host name portion of the URL can be specified as either a domain name or an IP address. Because an IPv6 address can contain colon characters, you must enclose the entire address in bracket [] characters. If this is not done, this method will return an error indicating the port number is invalid. For example, the URL **https://[2001:db8:0:0:1::128]/** uses an IPv6 host address and this would be considered valid. Without the brackets, this URL would not be accepted.

Important: The URL provided to this method will only be used to establish a connection with a server. This is a general purpose method which does not enable support for any particular application protocol and all implementation details are the responsibility of your application. If you require higher-level support for a specific Internet protocol, the SocketTools ActiveX Edition provides a comprehensive collection of higher-level controls which can be used to access those services.

If you use the **swOptionSecure** option to enable a secure connection, the connection will always use implicit TLS. This means a secure session will be initiated immediately after the socket connection has been established with the server. A common example of a service which uses implicit TLS is the HTTPS protocol. Another type of secure connection is one that uses explicit TLS. This is when the client establishes a normal (non-secure) connection with the server and then explicitly switches to using a secure connection, typically by sending a command. If the server you are connecting to requires explicit TLS, you should not specify the **dwOptionSecure** option. Instead, connect without this option and then set the **Secure** property to True when you are ready to initiate the TLS handshake.

See Also

[HostName Property](#), [KeepAlive Property](#), [NoDelay Property](#), [Options Property](#), [Secure Property](#), [Timeout Property](#), [Disconnect Method](#)

Disconnect Method

Terminate the connection with a remote host.

Syntax

object.Disconnect

Parameters

None.

Return Value

A value of zero is returned if the connection was terminated successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

This method terminates the network connection with the server.

See Also

[Connect Method](#)

Flush Method

Flush the contents of the send and receive socket buffers.

Syntax

object.Flush

Parameters

None.

Return Value

A value of zero is returned if the socket buffers were flushed successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Flush** method will flush any data waiting to be read or written to the remote host . It is important to note that this method is not similar to flushing data to a disk file; it does not ensure that a specific block of data has been written to the socket. For example, you should never call this function immediately after calling the **Write** or **WriteLine** methods, or prior to calling the **Disconnect** method.

An application should not use the **Flush** method under normal circumstances. This method is only to be used when the application needs to immediately return the socket to an inactive state with no pending data to be read or written. Calling this method may result in data loss and should only be used if you understand the implications of discarding any data which has been sent by the remote host.

See Also

[IsReadable Property](#), [IsWritable Property](#), [Read Method](#), [Write Method](#)

Initialize Method

Initialize the control and validate the runtime license key.

Syntax

```
object.Initialize( [LicenseKey] )
```

Parameters

LicenseKey

An optional string value which specifies a runtime license key used to initialize the control. If the license key is omitted or passed as an empty string, a development license must be installed on the local system.

Return Value

A value of zero is returned if the control was initialized successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

This method dynamically loads other system libraries and allocates thread local storage. In most cases, it is not necessary to call this method directly because it is automatically invoked when an instance of the control is created by the container. However, if the control is created dynamically using **CreateObject** or a similar method, this must be the first method that is called before you attempt to modify any property values or invoke other methods. Failure to initialize the control may result in subsequent errors and/or cause an exception to be raised.

Each time the **Initialize** method is invoked, it increments an internal counter that keeps track of the number of times that it has been called by any thread in the current process. The **Uninitialize** method decrements this counter, and when the usage count drops to zero, the control will automatically unload the system libraries that it has dynamically loaded and will destroy the process heap that was allocated when the first instance of the control was created.

If the control is being used within another DLL, it is important that you do not attempt to create an instance of the control or call the **Initialize** method from within the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is written in C++ and it is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for any static and global C++ objects and has the same restrictions.

Example

```
Set objSocket = CreateObject("SocketTools.SocketWrench.11")

nError = objSocket.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketWrench"
End
End If
```

See Also

[IsInitialized Property](#), [Connect Method](#), [Reset Method](#), [Uninitialize Method](#)

Listen Method

Listen for incoming connections.

Syntax

object.Listen([*LocalAddress*], [*LocalPort*], [*Backlog*])

Parameters

LocalAddress

An optional string value that specifies the IP address the control should use when listening for connection requests. If this argument is not specified, the control will bind to any suitable IPv4 interface on the local system.

LocalPort

An optional integer value that specifies the local port number that be used to listen for connections. If this parameter is omitted, the **LocalPort** property will be used to determine the default port number. If this value is zero, the listening socket will be bound to a random port number.

Backlog

An optional integer value that specifies the maximum size of the queue used to manage pending connections to the service. If this parameter is set to value which exceeds the maximum size allowed by the operating system, it will be silently adjusted to the nearest legal value. If this parameter is omitted, the **Backlog** property will be used to determine the default value.

Return Value

A value of zero is returned if the operation was successful, otherwise a non-zero error code is returned which indicates the cause of the failure. This method will return an error if a socket has already been created by a previous call to the **Connect** method.

Remarks

The **Listen** method causes the control to listen on a socket for incoming connections on a particular local address and local port. If an IPv6 address is specified as the local IP address, the system must have an IPv6 stack installed and configured, otherwise the method will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

See Also

[Backlog Property](#), [LocalPort Property](#), [Accept Method](#), [Connect Method](#), [Reject Method](#), [OnAccept Event](#)

Peek Method

Return data read from the socket, but do not remove it from the socket buffer.

Syntax

object.Peek(Buffer, [Length])

Parameters

Buffer

A buffer that the data will be stored in. If the variable is a **String** then the data will be returned as a string of characters. This is the most appropriate data type to use if the server is sending data that consists of printable characters. If the server is sending binary data, it is recommended that a **Byte** array be used instead. This parameter must be passed by reference.

Length

A numeric value which specifies the number of bytes to read. Its maximum value is $2^{31}-1 = 2147483647$. This argument is required to be present for string data. If a value is specified for this argument for other permissible types of data, and it is less than number of bytes that is determined by the control, then **Length** will override the internally computed value. If the argument is omitted, then the maximum number of bytes to read is determined by the size of the buffer.

Return Value

If the method succeeds, it will return the number of bytes available to read from the socket without causing the thread to block. A return value of zero indicates that there is no data available to read at that time. If an error occurs, a value of -1 is returned.

Remarks

The **Peek** method reads the specified number of bytes from the socket and copies them into the buffer, but it does not remove the data from the internal socket buffer. Note that it is possible for the returned data to contain embedded null characters.

The data returned by the **Peek** method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation, up to the specified buffer size. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. Using this method in a loop to poll a non-blocking socket may cause the application to become non-responsive. To determine if there is data available to be read, use the **IsReadable** property.

See Also

[IsReadable Property](#), [Read Method](#), [ReadLine Method](#), [Write Method](#), [WriteLine Method](#), [OnRead Event](#), [OnWrite Event](#)

Read Method

Return data read from the socket.

Syntax

object.Read(*Buffer*, [*Length*], [*Options*], [*RemoteAddress*], [*RemotePort*])

Parameters

Buffer

A buffer that the data will be stored in. If the variable is a **String** then the data will be returned as a string of characters. This is the most appropriate data type to use if the server is sending data that consists of printable characters. If the server is sending binary data, a **Byte** array should be used instead. This parameter must be passed by reference.

Length

An optional integer value which specifies the number of bytes to read. Its maximum value is $2^{31}-1 = 2147483647$. This argument is required to be present for string data. If a value is specified for this argument for other permissible types of data, and it is less than number of bytes that is determined by the control, then **Length** will override the internally computed value. If the argument is omitted, then the maximum number of bytes to read is determined by the size of the buffer.

Options

An optional integer value that is reserved for future functionality and should either be omitted, or specified with a value of zero. Specifying a non-zero value will cause the method to fail and return an error.

RemoteAddress

An optional string that will contain the IP address of the remote host when the method returns. This parameter must be passed by reference. For a TCP connection, the IP address is the same value that was used to establish the connection. When reading data from a UDP socket, this is the IP address of the peer that sent the datagram. This information can be used in conjunction with the **Write** method to send a datagram back to that host. If the peer's IP address is not required, this parameter may be omitted.

RemotePort

An optional integer that will contain the port number for the remote host when the method returns. This parameter must be passed by reference. When reading a datagram from a UDP socket, this is the port number used by the peer who sent the datagram. This information can be used in conjunction with the **Write** method to send a datagram back to that host. If the peer's port number is not required, this parameter may be omitted.

Return Value

The number of bytes actually read from the socket is returned by this method. If an error occurs, a value of -1 is returned.

Remarks

The **Read** method returns data that has been read from the socket, up to the number of bytes specified. If no data is available to be read, an error will be generated if the control is non-blocking mode. If the control is in blocking mode, the program will wait until data is returned by the server or the connection is closed.



If the data contains binary characters, particularly non-printable control characters and embedded nulls, you should always provide a **Byte** array to the **Read** method. When you provide a **String** variable as the buffer, the control will process the data as text. Binary characters may be interpreted as 8-bit ANSI encoding and embedded null characters will corrupt the data. Reading the data into a byte array ensures that you receive the data exactly as it was sent by the server.

If the remote host is sending text that you want to read a line at a time, use the **ReadLine** method. If you wish to read a large amount of data using a single method call rather than making multiple calls to the **Read** method, use the **ReadStream** method.

See Also

[CodePage Property](#), [IsReadable Property](#), [ReadLine Method](#), [ReadStream Method](#) [Write Method](#), [OnRead Event](#), [OnWrite Event](#)

ReadByte Method

Read a byte of data from the socket.

Syntax

object.ReadByte

Parameters

None.

Return Value

The integer value of the byte read from the socket. If an error occurs, the method will return a value of -1 and the program should check the value of the **LastError** property to determine the specific cause of the error.

Remarks

The **ReadByte** method returns one byte of data that has been read from the socket. If no data is available to be read, an error will be generated if the control is non-blocking mode. If the control is in blocking mode, the program will stop until a byte of data is returned by the server or the connection is closed.

Note that you should not use the **ReadByte** method with a datagram socket. If you do, then only the first byte of the datagram will be returned and the remaining data will be discarded. When reading data from a datagram socket, it is recommended that you always use the **Read** method with the length argument specifying the maximum size of the datagram.

See Also

[IsReadable Property](#), [Timeout Property](#), [Read Method](#), [Write Method](#), [WriteByte Method](#), [OnRead Event](#)

ReadLine Method

Read up to a line of data from the socket and returns it in a string buffer.

Syntax

*object.ReadLine(**Buffer**, [**Length**])*

Parameters

Buffer

A buffer that the data will be stored in. If the variable is a **String** then the data will be returned as a string of characters. If the data returned by the server contains UTF-8 encoded text, it will automatically be converted to standard UTF-16 Unicode text. If you wish to read the data without conversion, provide a **Byte** array as the buffer. This parameter must be passed by reference.

Length

A numeric value which specifies the number of bytes to read. Its maximum value is $2^{31}-1 = 2147483647$. This argument is required to be present for string data. If a value is specified for this argument for other permissible types of data, and it is less than number of bytes that is determined by the control, then **Length** will override the internally computed value. If the argument is omitted, then the maximum number of bytes to read is determined by the size of the buffer.

Return Value

This method will return True if a line of data has been read. If an error occurs or there is no more data available to read, then the method will return False. It is possible for data to be returned in the string buffer even if the return value is False. Applications should check the length of the string after the method returns to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the function return value.

Remarks

The **ReadLine** method reads data from the socket up to the specified number of bytes or until an end-of-line character sequence is encountered. Unlike the **Read** method which reads arbitrary bytes of data, this function is specifically designed to return a single line of text data in a string variable. When an end-of-line character sequence is encountered, the function will stop and return the data up to that point; the string will not contain the carriage-return or linefeed characters.

If multi-byte 8-bit encoded characters are read from the socket, by default they will automatically be converted to Unicode according to the value of the **CodePage** property and returned in the string buffer provided. To prevent the text from being converted to Unicode, call the **Read** method and use a byte array instead of a string variable.

There are some limitations when using the **ReadLine** method. The method should only be used to read text, never binary data. In particular, it will discard nulls, linefeed and carriage return control characters. This method will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection. If the **Blocking** property is set to False, calling this method will automatically switch the socket into a blocking mode, read the data and then restore the socket to non-blocking mode. If another socket operation is attempted while **ReadLine** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking socket connections.

The **Read** and **ReadLine** methods can be intermixed, however be aware that the **Read** method will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

See Also

[CodePage Property](#), [IsReadable Property](#), [Read Method](#), [ReadStream Method](#), [StoreStream Method](#), [Write Method](#), [WriteLine Method](#)

ReadStream Method

Read the socket and store the data stream in the specified buffer.

Syntax

object.ReadStream(*Buffer*, [*Length*], [*Marker*], [*Options*])

Parameters

Buffer

A variable that will contain the data read from the socket when the method returns. If the variable is a **String** type, then the data will be stored as a string of characters. This is the most appropriate data type to use if the server is sending text data that consists of printable characters. If the remote host is sending binary data, a **Byte** array should be used instead. This parameter must be passed by reference.

Length

A numeric variable which specifies the maximum amount of data to be read from the socket. When the method returns, this variable will be updated with the actual number of bytes read. Note that because this argument is passed by reference and modified by the method, you must provide a variable, not a numeric constant. If this argument is omitted or the value is initialized to zero, this method will read data from the socket until the remote host disconnects or an error occurs.

Marker

A string or array of bytes which is used to designate the logical end of the data stream. When this byte sequence is encountered by the method, it will stop reading and return to the caller. The buffer will contain all of the data read from the socket up to and including the end-of-stream marker. If this argument is omitted, then the function will continue to read from the socket until the maximum buffer size is reached, the remote host closes its socket or an error is encountered.

Options

An optional integer value which specifies any options to be used when reading the data stream. One or more of the following bit flags may be specified by the caller:

Value	Description
swStreamDefault	The data stream will be returned to the caller unmodified. This option should always be used with binary data or data being stored in a byte array. If no options are specified, this is the default option used by this method.
swStreamConvert	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.

Return Value

This method returns a boolean value. If the method succeeds, the return value is True. If the function fails, the return value is False. To get extended error information, check the value of the **LastError** property.

Remarks

The **ReadStream** method enables an application to read an arbitrarily large stream of data and store it in memory, either in a string or a byte array. Unlike the **Read** method, which will return immediately when any amount of data has been read, the **ReadStream** method will only return when the buffer is full as specified by the **Length** argument, the logical end-of-stream marker has been read, the socket closed by the remote host or when an error occurs.



If the data contains binary characters, particularly non-printable control characters and embedded nulls, you should always provide a **Byte** array to the **ReadStream** method. When you provide a **String** variable as the buffer, the control will process the data as text. Binary characters may be interpreted as 8-bit ANSI encoding and embedded null characters will corrupt the data. Reading the data into a byte array ensures that you receive the data exactly as it was sent by the server.

This method will force the application to wait until the operation completes. If this method is called and the **Blocking** property is set to False, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **ReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for data to be returned in the buffer even if the method returns False. Applications should also check the value of the **Length** argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Because **ReadStream** can potentially cause the application to block for long periods of time as the data stream is being read, the control will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a blocking operation inside the event handler.

Example

```
Dim strBuffer As String
Dim nLength As Long

nLength = 0 ' Read socket until connection is closed
If SocketWrench1.ReadStream(strBuffer, nLength, Options:=swStreamConvert) Then
    TextBox1.Text = strBuffer
End If
```

See Also

[Blocking Property](#), [CodePage Property](#), [Read Method](#), [ReadLine Method](#), [StoreStream Method](#), [WriteStream Method](#), [OnProgress Event](#)

Reject Method

Rejects a connection request from a remote host.

Syntax

object.Reject

Parameters

None.

Return Value

A value of zero is returned if the rejection was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Reject** method rejects a pending client connection and the remote host will see this as the connection being aborted. If there are no pending client connections at the time, this method will immediately return with an error indicating that the operation would cause the thread to block.

See Also

[Accept Method](#), [Listen Method](#), [OnAccept Event](#)

Reset Method

Reset the internal state of the control.

Syntax

object.Reset

Parameters

None.

Return Value

None.

Remarks

The **Reset** method resets the internal state of the control. Property values are initialized to their internal defaults, open network connections will be closed and any handles allocated by the control will be released.

See Also

[Cancel Method](#), [Initialize Method](#), [Uninitialize Method](#)

Resolve Method

Resolves a host name to a host IP address.

Syntax

object.Resolve(HostName, IpAddress)

Parameters

HostName

A string value that specifies the host name to resolve.

IpAddress

A string that will contain the IP address of the specified host when the method returns. This parameter must be passed by reference. The value that is returned may be either a dotted-quad IPv4 address or an IPv6 address, depending on the configuration of the local system and what addresses are assigned to the host name.

Return Value

A value of zero is returned if the host name could be resolved into an IP address. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Resolve** method is used to resolve a host name into an IP address. If the host name has both an IPv4 and IPv6 address associated with it, this method will return the IPv4 address by default. If the host name only has an IPv6 address, that value will be returned if the local system has an IPv6 TCP/IP stack installed; otherwise, the method will fail with an error indicating that the host name could not be resolved.

See Also

[AutoResolve Property](#), [HostAddress Property](#), [HostFile Property](#), [HostName Property](#)

Shutdown Method

Stop sending or receiving data on the socket.

Syntax

`object.Shutdown( [Option] )`

Parameters

Option

An optional integer value that specifies the action to be taken. It may be one of the following values:

Value	Description
swShutdownRead	Disable reception of data. The application will no longer be able to receive data from the remote host. The application may continue to send data using the Write or WriteLine method until the socket is closed.
swShutdownWrite	Disable transmission of data. The application will no longer be able to send data to the remote host and the remote host will consider the socket connection to be closed. The application may continue to read any remaining data in the socket's receive buffer using the Read or ReadLine method until the socket is closed. This is the default value if this parameter is omitted.
swShutdownBoth	Disable both reception and transmission of data. If this value is specified, then the socket handle remains valid, however the client will not be able to send or receive data. The application must call the Disconnect method to close the socket.

Return Value

A value of zero is returned if the request was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

In some asynchronous applications, it may be desirable for a client to inform the server that no further communication is wanted, while allowing the client to read any residual data that may reside in internal buffers on the client side. **Shutdown** accomplishes this because the socket handle is still valid after it has been called, although some or all communication with the remote host has ceased.

See Also

[Disconnect Method](#)

StoreStream Method

Reads the data stream from the socket and stores it in a specified file.

Syntax

object.StoreStream(FileName, [Length], [Offset], [Options])

Parameters

FileName

A string variable that specifies the name of the file that will contain the data read from the socket. If the file does not exist, it will be created. If the file does exist, it will be overwritten.

Length

A numeric variable which specifies the maximum amount of data to be read from the socket. When the method returns, this variable will be updated with the actual number of bytes read. Note that because this argument is passed by reference and modified by the method, you must provide a variable, not a numeric constant. If this argument is omitted or the value is initialized to zero, this method will read data from the socket until the remote host disconnects or an error occurs.

Offset

A numeric value which specifies the byte offset into the file where the method will start storing data read from the socket. Note that all data after this offset will be truncated. If this argument is omitted or a value of zero is specified the file will be completely overwritten if it already exists.

Options

An optional integer value which specifies any options to be used when reading the data stream. One or more of the following bit flags may be specified by the caller:

Value	Description
swStreamDefault	The data stream will be returned to the caller unmodified. This option should always be used with binary data or data being stored in a byte array. If no options are specified, this is the default option used by this method.
swStreamConvert	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.

Return Value

This method returns a boolean value. If the method succeeds, the return value is True. If the function fails, the return value is False. To get extended error information, check the value of the **LastError** property.

Remarks

The **StoreStream** method enables an application to read an arbitrarily large stream of data from the socket and store it in a file. This method is essentially a simplified version of the **ReadStream** method, designed specifically to be used with files rather than strings or byte arrays.

This method will force the thread to block until the operation completes. If this method is called with the **Blocking** property set to False, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **StoreStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

Because **StoreStream** can potentially cause the application to block for long periods of time as the data stream is being read, the control will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being read. Note that an application should never perform a blocking operation inside the event handler.

See Also

[Blocking Property](#), [ReadStream Method](#), [WriteStream Method](#), [OnProgress Event](#)

Uninitialize Method

Uninitialize the control and release any system resources that were allocated.

Syntax

object.Uninitialize

Parameters

None.

Return Value

None.

Remarks

The **Uninitialize** method terminates any connection established by the control and resets the internal state of the control. This method is not typically used because any resources that have been allocated by an instance of the control will automatically be released when it is destroyed.

Each time the **Initialize** method is invoked, it increments an internal counter that keeps track of the number of times that it has been called by any thread in the current process. The **Uninitialize** method decrements this counter, and when the usage count drops to zero, the control will automatically unload the system libraries that it has dynamically loaded and will destroy the process heap that was allocated when the first instance of the control was created. An application should only call the **Uninitialize** method if it has explicitly called the **Initialize** method.

See Also

[Initialize Method](#)

Write Method

Write data to the socket.

Syntax

object.Write(*Buffer*, [*Length*], [*Options*], [*RemoteAddress*], [*RemotePort*])

Parameters

Buffer

A buffer variable that contains the data to be written to the server. If the variable is a **String** type, then the data will be written as a string of characters. This is the most appropriate data type to use if the server expects text data that consists of printable characters. If the string contains Unicode characters, it will be automatically converted to use standard UTF-8 encoding prior to being sent. If the server is expecting binary data, a **Byte** array should be used instead.

Length

An optional integer value that specifies the maximum number of bytes to send to the server. Its maximum value is $2^{31}-1 = 2147483647$. This argument is not required for string data. If a value is specified for this argument for other permissible types of data, and it is less than number of bytes that is determined by the control, then **Length** will override the internally-computed value. If the socket is non-blocking and the send fails because it could not write all of the data to the server, the **OnWrite** event will be fired when the server can be written to again.

Options

An optional integer value that is reserved for future functionality and should either be omitted, or specified with a value of zero. Specifying a non-zero value will cause the method to fail and return an error.

RemoteAddress

An optional string value that specifies the IP address of the remote host that the data will be sent to. For a TCP connection, it is recommended that this argument be omitted. If it is specified, the IP address must be the same value that was used to establish the connection. When writing data to a UDP socket, this is the IP address of the peer that will receive the datagram. This information can be used in conjunction with the **Read** method to send a datagram back to that host.

RemotePort

An optional integer value that specifies the port number on the remote host that the data will be sent to. For a TCP connection, it is recommended that this argument be omitted. If it is specified, the port number must be the same value that was used to establish the connection. When writing data on a UDP socket, this is the port number for the peer who will receive the datagram. This information can be used in conjunction with the **Read** method to send a datagram back to that host.

Return Value

This method returns the number of bytes actually written to the socket, or -1 if an error was encountered.

Remarks

The **Write** method sends the data in *buffer* to the socket. If the connection is buffered, as is typically the case, the data is copied to the send buffer and control immediately returns to the program. If the control is non-blocking and is out of buffer space, an error will be generated. If the control is blocking,

the application will wait until the data can be sent.



If the data contains binary characters, particularly non-printable control characters and embedded nulls, you should always provide a **Byte** array to the **Write** method. When you provide a **String** variable as the buffer, the control will process the data as text. If the string contains non-ASCII characters, by default they will automatically be converted to 8-bit ANSI encoded text prior to being written. Using a byte array ensures that binary data will be sent as-is without being encoded.

If you want to send text to the remote host a line at a time, use the **WriteLine** method. If you wish to send a large amount of data using a single method call rather than making multiple calls to the **Write** method, use the **WriteStream** method.

See Also

[CodePage Property](#), [IsWritable Property](#), [Timeout Property](#), [Read Method](#), [WriteLine Method](#), [WriteStream Method](#), [OnWrite Event](#)

WriteByte Method

Write a byte of data to the socket.

Syntax

object.WriteByte(*Value*)

Parameters

Value

A byte or integer value that specifies the data that should be sent to the remote host. If this parameter is a numeric value, it will be converted to its equivalent byte value and written to the socket. If the argument is a string, the first character will be written to the socket.

Return Value

This method returns a boolean value. If the byte of data was successfully written to the socket, the method will return True. If the data could not be written to the socket, the method will return False and the application should check the value of the **LastError** property to determine the exact cause of the failure.

Remarks

The **WriteByte** method writes a single byte of data to the socket. If the connection is buffered, as is typically the case, the data is copied to the send buffer and control immediately returns to the program. If the socket is non-blocking and is out of buffer space, an error will be generated. If the socket is blocking, the application will wait until the data can be sent.

If you use the **WriteByte** method with a datagram socket, the datagram will only be a single byte in length. You cannot use multiple calls to **WriteByte** to compose a single datagram.

See Also

[IsWritable Property](#), [Timeout Property](#), [Read Method](#), [ReadByte Method](#), [Write Method](#), [OnWrite Event](#)

WriteLine Method

Send a line of text to the remote host, terminated by a carriage-return and linefeed.

Syntax

object.WriteLine([*Buffer*])

Parameters

Buffer

An optional **String** value which contains the text that will be sent to the remote host. The data will always be terminated with a carriage-return and linefeed control character sequence. If this argument is omitted, then a only a carriage-return and linefeed are written to the socket. If the string contains Unicode characters, it will be automatically converted to use standard UTF-8 encoding prior to being sent. If the string contains an embedded null character, any data that follows the null character will be discarded.

Return Value

This method returns True if the contents of the string have been written to the socket. If an error occurs, the method will return False.

Remarks

The **WriteLine** method writes a line of text to the remote host and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **Write** method which writes arbitrary bytes of data to the socket, this method is specifically designed to write a single line of text data from a string.

If the **Buffer** string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the number of characters sent to the remote host will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

If the string value passed to the **WriteLine** method is a Unicode string which contains non-ASCII characters, it will be internally converted to 8-bit ANSI encoded text before being written to the socket. The remote host must be able to recognize the encoding and process it appropriately. The **ReadLine** method will automatically convert any encoded characters that it reads from the socket back to their original Unicode encoding. The **CodePage** property can be used to change the default code page used when converting the text.

The **WriteLine** method should only be used to send text, never binary data. In particular, the function will discard any data that follows a null character and will append linefeed and carriage return control characters to the data stream. Calling this this method will force the thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection. If this function is called with the **Blocking** property set to False, it will automatically switch the socket into a blocking mode, send the data and then restore the socket to non-blocking mode. If another socket operation is attempted while the **WriteLine** method is blocked sending data to the remote host, an error will occur. It is recommended that this method only be used with blocking socket connections.

The **Write** and **WriteLine** function calls can be safely intermixed.

See Also

[CodePage Property](#), [IsWritable Property](#), [Timeout Property](#), [Read Method](#), [ReadLine Method](#), [Write Method](#), [WriteStream Method](#)

WriteStream Method

Writes data from the stream buffer to the socket.

Syntax

object.WriteStream(*Buffer*, [*Length*], [*Options*])

Parameters

Buffer

A variable that contains the data to be written to the socket. If the variable is a **String** type, then the data will be stored as a string of characters. This is the most appropriate data type to use if the server is expecting text data that consists of printable characters. If the string contains Unicode characters, it will be automatically converted to use standard UTF-8 encoding prior to being sent. If the server is expecting binary data, a **Byte** array should be used instead.

Length

A numeric variable which specifies the maximum amount of data to be written to the socket. When the method returns, this variable will be updated with the actual number of bytes written. Note that because this argument is passed by reference and modified by the method, you must provide a variable, not a numeric constant. If this argument is omitted or the value is initialized to zero, this method will automatically determine the amount of data based on the length of the string or the size of the byte array passed to the method.

Options

An optional integer value which specifies any options to be used when writing the data stream to the socket. Currently this argument is reserved for future expansion and should either be omitted or always specified with a value of zero.

Return Value

This method returns a boolean value. If the function succeeds, the return value is True. If the function fails, the return value is False. To get extended error information, check the value of the **LastError** property.

Remarks

The **WriteStream** method enables an application to write an arbitrarily large stream of data from a string buffer or byte array to the socket. Unlike the **Write** method, which may not write all of the data in a single call, the **WriteStream** method will only return when all of the data has been written or an error occurs.



If the data contains binary characters, particularly non-printable control characters and embedded nulls, you should always provide a **Byte** array to the **WriteStream** method. When you provide a **String** variable as the buffer, the control will process the data as text. If the string contains Unicode characters, they will automatically be converted to 8-bit ANSI encoded text prior to being written. Using a byte array ensures that binary data will be sent as-is without being encoded. You can change the default code page used to convert the text by setting the **CodePage** property.

This method will force the application to wait until the operation completes. If this method is called with the **Blocking** property set to False, it will automatically switch the socket into a blocking mode, write the data stream and then restore the socket to non-blocking mode when it has finished. If another socket operation is attempted while **WriteStream** is blocked sending data to the remote

host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible that some data will have been written to the socket even if the method returns False. Applications should also check the value of the **Length** argument to determine if any data was sent. For example, if a timeout occurs while the function is waiting to write more data, it will return zero; however, some data may have already been written to the socket prior to the error condition.

Because **WriteStream** can potentially cause the application to wait for long periods of time as the data stream is being written, the control will periodically generate **OnProgress** events. An application can use this event to update the user interface as the data is being written. Note that an application should never perform a blocking operation inside the event handler.

Example

```
Dim hFile As Long
Dim nLength As Long
Dim dataBuffer() As Byte
Dim bResult As boolean

' Open the file for binary access
hFile = FreeFile()
Open strFileName For Binary Access Read As hFile

' Determine the size of the file and allocate a byte
' array large enough to store the contents
nLength = LOF(hFile)
ReDim dataBuffer(nLength - 1) As Byte

' Read the file contents into the byte array and
' then close the file
Get hFile, , dataBuffer
Close hFile

' Write the data to the socket
bResult = SocketWrench1.WriteStream(dataBuffer, nLength)
```

See Also

[Blocking Property](#), [CodePage Property](#), [ReadStream Method](#), [StoreStream Method](#), [Write Method](#), [OnProgress Event](#)

SocketWrench Control Events

Event	Description
OnAccept	This event occurs when a remote host connects to a listening socket
OnCancel	This event occurs when a blocking operation is canceled
OnConnect	This event occurs when a connection is established
OnDisconnect	This event occurs when a connection is terminated
OnError	This event occurs when a control error occurs
OnProgress	This event occurs as a data stream is being read or written
OnRead	This event occurs when data is available to be read
OnTimeout	This event occurs when a blocking operation times out
OnTimer	This event occurs when the control's preset timer interval expires
OnWrite	This event occurs when data can be written to the server

OnAccept Event

The **OnAccept** event is generated when a remote host connects to a listening socket.

Syntax

Sub *object_OnAccept* ([*Index As Integer*,] **ByVal** *Handle As Variant*)

Remarks

This event occurs for sockets that are listening for connections from a remote host. A connection with the remote system is not actually established until it has been accepted by the listening server. This event is only generated for asynchronous sockets when the **Blocking** property is set to False.

The **Handle** argument specifies the socket descriptor of the listening socket. To accept the connection, a socket calls its **Accept** method with argument **Handle**.

The **PeerAddress** or **PeerName** properties may be used to determine the name of the remote host that is establishing the connection. Note that this information may not be available until after the **Accept** method is called to accept the connection.

See Also

[PeerAddress Property](#), [PeerName Property](#), [Accept Method](#), [Reject Method](#)

OnCancel Event

The **OnCancel** event is generated when a blocking operation is canceled.

Syntax

Sub *object_OnCancel* ([*Index As Integer*])

Remarks

This event occurs when a blocking operation on the socket, such as sending or receiving data, is canceled with the **Cancel** method.

To assist in determining which operation was canceled, consult the **State** property.

See Also

[Cancel Method](#), [OnError Event](#)

OnConnect Event

The **OnConnect** event is generated when a connection is established.

Syntax

Sub *object_OnConnect* ([*Index As Integer*])

Remarks

The **OnConnect** event is generated when a connection is made with a remote host as a result of a **Connect** method call, or when an **Accept** method call is completed. This event is only generated for asynchronous sockets when the **Blocking** property is set to False.

See Also

[Accept Method](#), [Connect Method](#), [OnDisconnect Event](#)

OnDisconnect Event

The **OnDisconnect** event is generated when a connection is terminated.

Syntax

Sub *object_OnDisconnect* ([*Index As Integer*])

Remarks

The **OnDisconnect** event is generated when the connection is terminated by the remote host. This event is only generated for asynchronous sockets when the **Blocking** property is set to False.

See Also

[OnConnect Event](#)

OnError Event

The **OnError** event is generated when a control error occurs.

Syntax

```
Sub object_OnError ( [Index As Integer,] ByVal ErrorCode As Variant, ByVal Description As Variant )
```

Remarks

This event occurs when an error occurs during a control action. Visual Basic errors do not generate this event.

The ***ErrorCode*** argument specifies the last error that has occurred. If the error is network related, the error code values returned by the control correspond to those returned by the standard Windows Sockets library.

The ***Description*** argument is a string that describes the error.

See Also

[LastError Property](#), [LastErrorString Property](#), [ThrowError Property](#)

OnProgress Event

The **OnProgress** event is generated as the data stream is being read or written.

Syntax

Sub *object_OnProgress* ([*Index As Integer*], *ByVal BytesTotal As Variant*, *ByVal BytesCopied As Variant*, *ByVal Percent As Variant*)

Remarks

The **OnProgress** event is generated as the control reads the data stream from the remote host or writes a data stream to the remote host. If the data stream contains large amounts of data, this event can be used to update a progress bar or other user-interface control to provide the user with some visual feedback. The arguments to this event are:

BytesTotal

The total amount of data being read or written in bytes. This value will be the same as the maximum size of the data stream specified by the caller. If the size was unknown or unspecified at the time, then this value will always be the same as the *BytesCopied* value.

BytesCopied

The number of bytes that have been read or written.

Percent

The percentage of data that's been read or written, expressed as an integer value between 0 and 100, inclusive. If the maximum size of the data stream was not specified by the caller, this value will always be 100.

Note that this event is only generated by the **ReadStream**, **StoreStream** and **WriteStream** methods. If the control is reading or writing data using the **Read** or **Write** methods the application is responsible for calculating the completion percentage and updating any user interface controls.

See Also

[Read Method](#), [ReadStream Method](#), [StoreStream Method](#), [Write Method](#), [WriteStream Method](#)

OnRead Event

The **OnRead** event is generated when data is available to be read.

Syntax

Sub *object_OnRead* ([*Index As Integer*])

Remarks

The **OnRead** event is generated for non-blocking sockets when data is available to be read from the server. Use the **Read** method to read the data. This event is only generated for asynchronous sockets when the **Blocking** property is set to False.

See Also

[IsReadable Property](#), [Peek Method](#), [Read Method](#), [Write Method](#), [OnWrite Event](#)

OnTimeout Event

The **OnTimeout** event is fired when a blocking operation times out.

Syntax

Sub *object_OnTimeout* ([*Index As Integer*])

Remarks

The **OnTimeout** event is generated when a blocking socket operation, such as sending or receiving data, times out. To determine which operation was in progress when the timeout occurred, consult the **State** property.

See Also

[Timeout Property](#), [OnCancel Event](#)

OnTimer Event

The **OnTimer** event is fired when the control's preset timer interval expires.

Syntax

Sub *object_OnTimer* ([*Index As Integer*])

Remarks

This event occurs when the control's timer interval has elapsed. The frequency is specified in milliseconds by setting the **Interval** property.

See Also

[Interval Property](#)

OnWrite Event

The **OnWrite** event is generated when data can be written to the server.

Syntax

Sub *object_OnWrite* ([*Index As Integer*])

Remarks

The **OnWrite** event is generated for non-blocking sockets when data can be written to the server after a previous attempt failed because it would cause the control to block. This event is only generated for asynchronous sockets when the **Blocking** property is set to False.

This event will always be generated at least one time, after the connection to the server is initially established. It will not fire again unless the **Write** method fails with the error **swErrorOperationWouldBlock**, which indicates that the socket's send buffer is full. When the socket can accept more data, this event will fire and the application can resume sending data to the remote host.

See Also

[IsWritable Property](#), [Read Method](#), [Write Method](#), [OnRead Event](#)

SocketWrench Control Error Codes

Value	Constant	Description
10001	swErrorNotHandleOwner	Handle not owned by the current thread
10002	swErrorFileNotFound	The specified file or directory does not exist
10003	swErrorFileNotCreated	The specified file could not be created
10004	swErrorOperationCanceled	The blocking operation has been canceled
10005	swErrorInvalidFileType	The specified file is a block or character device, not a regular file
10006	swErrorInvalidDevice	The specified device or address does not exist
10007	swErrorTooManyParameters	The maximum number of function parameters has been exceeded
10008	swErrorInvalidFileName	The specified file name contains invalid characters or is too long
10009	swErrorInvalidFileHandle	Invalid file handle passed to function
10010	swErrorFileReadFailed	Unable to read data from the specified file
10011	swErrorFileWriteFailed	Unable to write data to the specified file
10012	swErrorOutOfMemory	Out of memory
10013	swErrorAccessDenied	Access denied
10014	swErrorInvalidParameter	Invalid argument passed to function
10015	swErrorClipboardUnavailable	The system clipboard is currently unavailable
10016	swErrorClipboardEmpty	The system clipboard is empty or does not contain any text data
10017	swErrorFileEmpty	The specified file does not contain any data
10018	swErrorFileExists	The specified file already exists
10019	swErrorEndOfFile	End of file
10020	swErrorDeviceNotFound	The specified device could not be found
10021	swErrorDirectoryNotFound	The specified directory could not be found
10022	swErrorInvalidBuffer	Invalid memory address passed to function
10024	swErrorNoHandles	No more handles available to this process
10035	swErrorOperationWouldBlock	The specified operation would block the current thread
10036	swErrorOperationInProgress	A blocking operation is currently in progress
10037	swErrorAlreadyInProgress	The specified operation is already in progress
10038	swErrorInvalidHandle	Invalid handle passed to function
10039	swErrorInvalidAddress	Invalid network address specified
10040	swErrorInvalidSize	Datagram is too large to fit in specified buffer
10041	swErrorInvalidProtocol	Invalid network protocol specified
10042	swErrorProtocolNotAvailable	The specified network protocol is not available
10043	swErrorProtocolNotSupported	The specified protocol is not supported

10044	swErrorSocketNotSupported	The specified socket type is not supported
10045	swErrorInvalidOption	The specified option is invalid
10046	swErrorProtocolFamily	The specified protocol family is not supported
10047	swErrorProtocolAddress	The specified address is invalid for this protocol family
10048	swErrorAddressInUse	The specified address is in use by another process
10049	swErrorAddressUnavailable	The specified address cannot be assigned
10050	swErrorNetworkUnavailable	The networking subsystem is unavailable
10051	swErrorNetworkUnreachable	The specified network is unreachable
10052	swErrorNetworkReset	Network dropped connection on reset
10053	swErrorConnectionAborted	Connection was aborted due to timeout or other failure
10054	swErrorConnectionReset	Connection was reset by remote network
10055	swErrorOutOfBuffers	No buffer space is available
10056	swErrorAlreadyConnected	Connection already established with remote host
10057	swErrorNotConnected	No connection established with remote host
10058	swErrorConnectionShutdown	Unable to send or receive data after connection shutdown
10060	swErrorOperationTimeout	The specified operation has timed out
10061	swErrorConnectionRefused	The connection has been refused by the remote host
10064	swErrorHostUnavailable	The specified host is unavailable
10065	swErrorHostUnreachable	The specified host is unreachable
10067	swErrorTooManyProcesses	Too many processes are using the networking subsystem
10091	swErrorNetworkNotReady	Network subsystem is not ready for communication
10092	swErrorInvalidVersion	This version of the operating system is not supported
10093	swErrorNetworkNotInitialized	The networking subsystem has not been initialized
10101	swErrorRemoteShutdown	The remote host has initiated a graceful shutdown sequence
11001	swErrorInvalidHostName	The specified hostname is invalid or could not be resolved
11002	swErrorHostNameNotFound	The specified hostname could not be found
11003	swErrorHostNameRefused	Unable to resolve hostname, request refused
11004	swErrorHostNameNotResolved	Unable to resolve hostname, no address for specified host
12001	swErrorInvalidLicense	The license for this product is invalid
12002	swErrorProductNotLicensed	This product is not licensed to perform this operation
12003	swErrorNotImplemented	This function has not been implemented on this platform
12004	swErrorUnknownLocalHost	Unable to determine local host name
12005	swErrorInvalidHostAddress	Invalid host address specified
12006	swErrorInvalidServicePort	Invalid service port number specified
12007	swErrorInvalidServiceName	Invalid or unknown service name specified
12008	swErrorInvalidEventId	Invalid event identifier specified

12009	swErrorOperationNotBlocking	No blocking operation in progress on this socket
12101	swErrorSecurityNotInitialized	Unable to initialize security interface for this process
12102	swErrorSecurityContext	Unable to establish security context for this session
12103	swErrorSecurityCredentials	Unable to open client certificate store or establish client credentials
12104	swErrorSecurityCertificate	Unable to validate the certificate chain for this session
12105	swErrorSecurityDecryption	Unable to decrypt data stream
12106	swErrorSecurityEncryption	Unable to encrypt data stream
12337	swErrorMaximumConnections	The maximum number of client connections exceeded
12338	swErrorThreadCreationFailed	Unable to create a new thread for the current process
12339	swErrorInvalidThreadHandle	The specified thread handle is no longer valid
12340	swErrorThreadTerminated	The specified thread has been terminated
12341	swErrorThreadDeadlock	The operation would result in the current thread becoming deadlocked
12342	swErrorInvalidClientMoniker	The specified moniker is not associated with any client session
12343	swErrorClientMonikerExists	The specified moniker has been assigned to another client session
12344	swErrorServerInactive	The specified server is not listening for client connections
12345	swErrorServerSuspended	The specified server is suspended and not accepting client connections

Internet Server Class Library

A general purpose TCP/IP networking library for developing server applications.

Reference

- [Data Members](#)
- [Class Methods](#)
- [Event Handlers](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CInternetServer
File Name	CSWSKV12.DLL
Version	12.0.3024.1286
LibID	99EEE601-E935-4266-B046-771A823DB47D
Import Library	CSWSKV12.LIB
Dependencies	None
Standards	RFC 768, RFC 791, RFC 793

Overview

The Internet Server class library provides a simplified interface for creating event-driven, multithreaded server applications using the TCP/IP protocol. Each instance of the Internet Server class represents a server, and each active client connection is managed internally and referenced by a handle which uniquely identifies the client session. The class library supports secure connections using the standard TLS protocol and can be used to create secure, custom server programs.

This class is designed to be used as a base class from which your own server class is derived. To exchange data with the clients that connect to the server, you should override the default events such as **OnConnect** and **OnRead**. Most interaction with the clients occur within these event handlers. Because the client sessions are managed in worker threads that are separate from the main UI thread of your application, you may perform a blocking operation in response to an event without affecting the other clients that are connected to the server.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must

have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

CInternetServer Public Data Members

Member Variables	Description
m_nBacklog	The size of the backlog connection queue for the server
m_nMaxClients	The maximum number of active client sessions accepted by the server
m_nMaxClientsPerAddress	The maximum number of clients per IP address accepted by the server
m_dwOptions	The options specified when creating an instance of the server
m_nPriority	The priority specified when creating an instance of the server
m_dwStackSize	The initial size of the stack allocated for threads created by the server
m_nTimeout	The timeout period in seconds waiting for a blocking operation to complete

CInternetServer::m_nBacklog

UINT m_nBacklog;

The size of the backlog connection queue for the server.

Remarks

The **m_nBacklog** data member is a public variable that specifies the size of the queue allocated for pending client connections. A value of zero specifies that the queue should be set to a reasonable default value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CInternetServer](#)

CInternetServer::m_nMaxClients

UINT m_nMaxClients;

The maximum number of clients that are permitted to connect to the server.

Remarks

The **m_nMaxClients** data member is a public variable that specifies the maximum number of clients that are permitted to establish a connection with the server. After this limit is reached, the server will reject additional connections until the number of active clients drops below this threshold. A value of zero specifies that there is no fixed limit on the active number of client connections. Changing the value of this data member does not have an effect on an active instance of the server. To change the maximum number of clients on an active server, use the **Throttle** method.

The actual number of client connections that can be accepted depends on the amount of memory available to the server process. Sockets are allocated from the non-paged memory pool, so the actual number of sockets that can be created system-wide depends on the amount of physical memory that is installed. If the server will be accessible over the Internet, it is recommended that you limit the maximum number of client connections to a reasonable value.

See Also

[CInternetServer](#), [Throttle](#)

CInternetServer::m_nMaxClientsPerAddress

UINT m_nMaxClientsPerAddress;

The maximum number of clients that are permitted to connect to the server from a single IP address.

Remarks

The **m_nMaxClientsPerAddress** data member is a public variable that specifies the maximum number of clients that are permitted to establish a connection with the server from a single IP address. After this limit is reached, the server will reject additional connections until the number of active clients drops below this threshold. A value of zero specifies that there is no limit on the active number of client connections per IP address. Changing the value of this data member does not have an effect on an active instance of the server. To change the maximum number of clients on an active server, use the **Throttle** method.

See Also

[CInternetServer](#), [Throttle](#)

CInternetServer::m_dwOptions

DWORD m_dwOptions;

The default options used when starting an instance of the server.

Remarks

The **m_dwOptions** data member is a public variable that specifies the default options that should be used when starting an instance of the server. This variable can be modified directly or by calling the **SetOptions** method. For a list of available server options, see [Server Option Constants](#). Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CInternetServer](#), [GetOptions](#), [SetOptions](#)

CInternetServer::m_nPriority

INT m_nPriority;

The priority specified when creating an instance of the server.

Remarks

The **m_nPriority** data member is a public variable that specifies the which specifies the priority for the server and all client sessions. Changing the value of this data member does not have an effect on an active instance of the server. It may be one of the following values:

Value	Description
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. It is recommended that most applications use this priority.
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the client session. It is typically used with lightweight services running in the background that are designed for few client connections. The client thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the client session and meters the processor utilization for the client session. The client thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the client session and the thread will be given higher scheduling priority. It can be used when it is important for the client session thread to be highly responsive. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The thread will be given higher scheduling priority and will be more responsive to the remote host. It is not recommended that this priority be used on a system with a single processor.

See Also

[CInternetServer](#), [GetPriority](#), [SetPriority](#)

CInternetServer::m_dwStackSize

DWORD m_dwStackSize;

The initial size of the stack allocated for threads created by the server.

Remarks

The **m_dwStackSize** data member is a public variable that specifies the initial amount of memory that is committed to the stack for each thread created by the server. A value of zero specifies that the default stack size should be used, which is 256K for 32-bit processes and 512K for 64-bit processes. This variable can be modified directly or by calling the **SetStackSize** method. Changing the value of this data member does not have an effect on an active instance of the server. It is recommended that most applications use the default stack size.

See Also

[CInternetServer](#), [GetStackSize](#), [SetStackSize](#)

CInternetServer::m_nTimeout

DWORD m_nTimeout;

The default options used when starting an instance of the server.

Remarks

The **m_nTimeout** data member is a public variable that specifies the number of seconds the server should wait for a client to perform a network operation. If the client does not exchange any information with the server within this period of time, a timeout event will occur. The timeout value affects all clients that are connected to the server. This variable can be modified directly or by calling the **SetTimeout** method. Changing the value of this data member does not have an effect on an active instance of the server.

See Also

[CInternetServer](#), [GetTimeout](#), [SetTimeout](#)

CInternetServer Class Methods

Class	Description
CInternetServer	Constructor which initializes the current instance of the class
~CInternetServer	Destructor which releases resources allocated by the class
Method	Description
Abort	Abort the connection and immediately close the socket
AsyncNotify	Enable or disable asynchronous notification of changes in server status
AttachHandle	Attach the specified server handle to this instance of the class
Broadcast	Write data to all active clients currently connected to the server
Cancel	Cancel a blocking operation for the specified client session
CompareAddress	Compare two IP addresses to determine if they are identical
DetachHandle	Detach the server handle from the current instance of this class
DisableSecurity	Disable secure communication with the client
DisableTrace	Disable logging of network function calls
Disconnect	Disconnect the client, closing the socket handle and terminating the session
EnableSecurity	Enable secure communication with the client
EnableTrace	Enable logging of network function calls to a file
EnumClients	Returns a list of active client connections established with the server
EnumNetworkAddresses	Return the list of network addresses that are configured for the local host
FindClient	Returns a handle to the client which matches the specified client ID or moniker
FormatAddress	Convert an IP address in binary format into a printable string
Flush	Flush the send and receive buffers for the specified client session
GetActiveClient	Return the socket handle for the active client session
GetAdapterAddress	Return the IP or MAC assigned to the specified network adapter
GetAddress	Convert an IP address string to a binary format
GetAddressFamily	Return the address family for the specified IP address
GetBacklog	Return the size of the backlog connection queue for the server
GetClientAddress	Return the IP address and port number for the specified client session
GetClientData	Returns the application defined data associated with the specified client session
GetClientHandle	Returns the handle for a specific client session based on its ID number
GetClientId	Returns the unique ID number assigned to the specified client session
GetClientIdleTime	Returns the amount of time the specified client session has been idle
GetClientMoniker	Returns the string alias associated with the specified client session
GetClientPort	Returns the remote port number used by the client to establish the connection
GetClientServer	Returns a socket handle to the server for the specified client socket
GetClientServerById	Returns a socket handle to the server for the specified session identifier

GetClientThreadId	Returns the thread ID for the specified client
GetClientThreads	Returns the number of client session threads created by the server
GetErrorString	Return a description for the specified error code
GetExternalAddress	Return the external IP address assigned to the local system
GetHandle	Return the client handle used by this instance of the class
GetHostAddress	Return the IP address assigned to the specified hostname
GetHostName	Return the hostname assigned to the specified IP address
GetLastError	Return the last error code
GetLocalAddress	Return the local IP address and port number for the server
GetLocalName	Return the hostname assigned to the local system
GetMemoryUsage	Return the amount of memory allocated for the server
GetOptions	Return the current server options
GetPriority	Return the current priority assigned to the server
GetStackSize	Return the initial size of the stack allocated for threads created by the server
GetStatus	Return the current status of the server
GetStreamInfo	Return information about the current stream I/O operation
GetThreadClient	Return the handle for the client session that is being managed by the specified thread
GetTimeout	Return the timeout interval for blocking operations in seconds
IsActive	Determine if the server is currently active
IsAddressNull	Determine if the specified IP address is a null address
IsAddressRoutable	Determine if the specified IP address is routable over the Internet
IsInitialized	Determine if the class has been successfully initialized
IsListening	Determine if the server is listening for client connections
IsLocked	Determine if the server is currently in a locked state
IsProtocolAvailable	Determine if the specified protocol and address family are supported
IsReadable	Determine if data is available to be read from the client
IsWritable	Determine if data can be sent to the client without causing the thread to block
Lock	Lock the server, causing all other client threads to block until it is unlocked
MatchHostName	Match a host name against one more strings that may contain wildcards
Peek	Read data from the client without removing it from the socket buffer
Read	Read data from the client
ReadLine	Read a line of data from the client, storing it in a string buffer
ReadStream	Read a stream of data from the client and store it in the specified buffer
Reject	Reject a pending client connection
Restart	Restart the server, terminating all active client sessions
Resume	Resume accepting client connections on the specified server
SetBacklog	Set the size of the backlog connection queue for the server

SetCertificate	Specify the server certificate that should be used with secure connections
SetClientData	Associate application defined data with the specified client session
SetClientMoniker	Associate a unique string alias with the specified client session
SetLastError	Set the last error code
SetOptions	Set one or more server options
SetPriority	Set the priority assigned to the server
SetTimeout	Set the timeout interval used when waiting for a blocking operation to complete
ShowError	Display a message box with a description of the specified error
Start	Begin listening for client connections on the specified address and port
Stop	Stop listening for connections and terminate all client sessions
StoreStream	Read a stream of data from the client and store it in a file
Suspend	Suspend accepting client connections and optionally reject or disconnect clients
Throttle	Limit the number of active client connections, connections per address and connection rate
Unlock	Unlock the server, allowing other client threads to resume execution
ValidateCertificate	Validate the specified security certificate is installed on the local system
Write	Write data to the client
WriteLine	Write a line of data to the client, terminated with a carriage-return and linefeed
WriteStream	Write a stream of data to the client

CInternetServer::CInternetServer Method

CInternetServer();

The **CInternetServer** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **InetInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[~CInternetServer](#), [IsInitialized](#)

CInternetServer::~CInternetServer

`~CInternetServer();`

The **CInternetServer** destructor releases resources allocated by the current instance of the **CInternetServer** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CInternetServer** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the connection are destroyed. If there are any clients connected to the server at the time the destructor is called, those client sessions will be immediately terminated.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSKV12.lib

See Also

[CInternetServer](#)

CInternetServer::Abort Method

```
BOOL Abort(  
    SOCKET hSocket  
);  
  
BOOL Abort();
```

Immediately close the socket without waiting for any remaining data to be written out.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Abort** method should only be used when the connection must be closed immediately. This method should only be used to abort client connections and should not be called within an **OnAccept** event handler. To reject an incoming client connection, use the **Reject** method.

In most cases, the server should call the **Disconnect** method to gracefully close a client connection. Aborting the connection will discard any buffered data and may cause errors or result in unpredictable behavior by the client application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[Cancel](#), [Disconnect](#), [Reject](#)

CInternetServer::AsyncNotify Method

```
BOOL AsyncNotify(  
    HWND hWnd,  
    UINT uMsg  
);
```

Enable or disable asynchronous notification of changes in server status.

Parameters

- hWnd**
A handle to the window whose window procedure will receive the notification message.
- uMsg**
The user-defined message that will be sent to the notification window.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **AsyncNotify** method is used by an application to enable or disable asynchronous notifications. The message window is typically the main UI window and these notifications are used signal to the application that it should update the user interface. If the **hWnd** parameter is not NULL, it must specify a valid window handle and the user-defined message must have a value of **WM_USER** or higher. The application cannot specify a notification message that is reserved by the operating system. The pseudo-handle **HWND_BROADCAST** cannot be specified as the notification window. If the **hWnd** parameter is NULL, notifications for the specified server will be disabled.

When asynchronous notifications are enabled for a server, the server will post the user-defined message to the window whenever there is a change in status or after a client has connected or disconnected from the server. The **wParam** message parameter will contain the notification message and the **lParam** message parameter will contain the handle to the server or the client ID. The following notification messages are defined:

Value	Description
INET_NOTIFY_STARTUP	This notification is sent when the server has started and is preparing to accept client connections. This notification is only sent once, and only if asynchronous notifications are enabled immediately after the Start method is called. This message will not be sent once the server has begun accepting client connections or when notification messages are disabled and then subsequently re-enabled at a later time. The lParam message parameter will specify the handle to the server.
INET_NOTIFY_LISTEN	This notification is sent when the server is listening for client connections. This notification message may be sent to the application multiple times over the lifetime of the server. If the server was suspended, this notification will be sent after the application calls the Resume method to resume

	accepting client connections. The <i>lParam</i> message parameter will specify the handle to the server.
INET_NOTIFY_SUSPEND	This notification is sent when the server suspends accepting new connections because the application has called the Suspend method. This notification message may be sent to the application multiple times over the lifetime of the server. The <i>lParam</i> message parameter will specify the handle to the server.
INET_NOTIFY_RESTART	This notification is sent when the server is restarted using the Restart method. Note that the server socket handle provided by the <i>lParam</i> message parameter will specify the new socket handle of the restarted server instance, not the original socket handle. The <i>lParam</i> message parameter will specify the handle to the server.
INET_NOTIFY_CONNECT	This notification is sent when the server accepts a client connection and the thread that manages the client session has begun processing network events for that client. This message notification will not be sent if the client connection is rejected by the server. The <i>lParam</i> message parameter will specify the unique ID of the client that connected to the server.
INET_NOTIFY_DISCONNECT	This notification is sent when the client disconnects from the server and the client socket has been closed. This notification message may not occur for each client session that is forced to terminate as the result of the server being stopped using the Stop method. The <i>lParam</i> message parameter will specify the unique ID of the client that disconnected from the server.
INET_NOTIFY_SHUTDOWN	This notification is sent when the server thread is in the process of terminating. At the time the application processes this notification message, the server handle in <i>lParam</i> will reference the defunct server and cannot be used with other server methods. The <i>lParam</i> message parameter will specify the handle to the server.

If asynchronous notifications are enabled, you should never use those notifications as a replacement for an event handler. When an event occurs, the callback function that handles the event is invoked in the context of the thread that manages the client session. The application should exchange data with the client within that event handler and not in response to a notification message. These notification messages should only be used to update the application UI in response to changes in the status of the server.

The INET_NOTIFY_CONNECT and INET_NOTIFY_DISCONNECT notifications are different from the other server notifications because the *lParam* message parameter does not specify the server handle, but rather the unique client ID associated with the session that connected to or disconnected from the server. If you need to obtain the handle to the client session using the ID, call the **GetClientHandle** method. To obtain the server handle in response to the INET_NOTIFY_CONNECT message, use the **GetClientServerById** method. Note that at the time the application processes the INET_NOTIFY_DISCONNECT notification message, the client session will have already terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetClientHandle](#), [GetClientServerById](#)

CInternetServer::AttachHandle Method

```
VOID AttachHandle(  
    SOCKET hSocket  
);
```

Attach the specified server socket handle to the current instance of the class.

Parameters

hSocket

The socket handle to the server that will be attached to the current instance of the class object.

Return Value

None.

Remarks

This method is used to attach a server handle created outside of the class using the SocketWrench API. Once the client handle is attached to the class, the other class member functions may be used with that server.

If a server handle has already been created for the class, that handle will be released when the new handle is attached to the class object. This will cause the server to stop and all client sessions will be terminated immediately. If you want to prevent the previous server from being stopped, you must call the **DetachHandle** method prior to attaching a new handle to the class instance.

Note that the *hSocket* parameter is presumed to be a valid server socket handle and no checks are performed to ensure that the handle references an active server. Specifying an invalid socket handle will cause subsequent method calls to fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[DetachHandle](#), [GetHandle](#)

CInternetServer::Broadcast Method

```
INT Broadcast(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

Sends data to all clients that are connected to the server.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server clients.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the method succeeds, the return value is the number of clients that the data was sent to. If the method fails, the return value is INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **Broadcast** method sends the contents of the buffer to all of the clients that are connected to the server. This method will block until all clients have been sent a copy of the data. There is no guarantee in which order the clients will receive and process the data that has been broadcast.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[Write](#), [WriteLine](#)

InternetServer::Cancel Method

```
BOOL Cancel(  
    SOCKET hSocket  
);
```

Cancel a blocking operation for the specified client session.

Parameters

hSocket

The handle to a client socket.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation using the same client socket handle. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

Canceling a blocking operation for another client session may yield unpredictable results. If you wish to terminate the client session, it is preferable to use the **Disconnect** method rather than using this method in conjunction with the **Abort** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[Abort](#), [Disconnect](#)

CInternetServer::CompareAddress Method

```
BOOL CompareAddress(  
    LPINTERNET_ADDRESS LpAddress1,  
    LPINTERNET_ADDRESS LpAddress2  
);  
  
BOOL CompareAddress(  
    LPCTSTR LpszAddress1,  
    LPCTSTR LpszAddress2  
);
```

Compare two IP addresses to determine if they are identical.

Parameters

LpAddress1

A pointer to an INTERNET_ADDRESS structure that contains the first IP address to be compared. An alternate version of this method accepts a string that specifies the IP address to be compared.

LpAddress2

A pointer to an INTERNET_ADDRESS structure that contains the second IP address to be compared. An alternate version of this method accepts a string that specifies the IP address to be compared.

Return Value

If the method succeeds and the two addresses are identical, the return value is non-zero. If the method fails or the two addresses are not identical, the return value is zero. If either parameter is NULL, or the address family for the two addresses are not the same, the last error code will be updated. If the addresses are valid and in the same address family, but are not identical, the last error code will be set to NO_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetClientAddress](#), [INTERNET_ADDRESS](#)

CInternetServer::DetachHandle Method

SOCKET DetachHandle();

The **DetachHandle** method detaches the server socket handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the server socket associated with the current instance of the class object. If the server is not active, the value `INVALID_SOCKET` will be returned.

Remarks

This method is used to detach a server handle created by the class for use with the SocketWrench API. Once the server handle is detached from the class, no other class member functions may be called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CInternetServer::DisableSecurity Method

```
BOOL DisableSecurity(  
    SOCKET hSocket  
);  
  
BOOL DisableSecurity();
```

Disable secure communication with the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DisableSecurity** method disables a secure session, with subsequent calls to **Read** and **Write** sending and receiving unencrypted data. It is important to note that because this method sends a shutdown message to terminate the secure session, this may cause connection to be closed by the remote host.

This method does not close the socket. Use the **Disconnect** method to close the socket and release the resources allocated for the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[EnableSecurity](#), [SetCertificate](#)

CInternetServer::DisableTrace Method

BOOL DisableTrace();

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cssock12.lib

See Also

[EnableTrace](#)

CInternetServer::Disconnect Method

```
BOOL Disconnect(  
    SOCKET hSocket  
);  
  
BOOL Disconnect();
```

Disconnect the client, closing the socket handle and terminating the session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

Once the connection has been terminated, the client socket handle is no longer valid and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new connection is established. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

This method sends an internal control message that notifies the server that this session should be terminated. When the session thread is signaled that it should terminate, it will begin to release the resources allocated for that session. To ensure that the client session terminates gracefully, there may be a brief period of time where the session thread is still active after this method has returned.

The **Disconnect** method should only be used to terminate client sessions and the server handle should never be provided as the *hSocket* parameter. To stop the server, use the **Stop** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[Abort](#), [Stop](#)

CInternetServer::EnableSecurity Method

```
BOOL EnableSecurity(  
    SOCKET hSocket  
);  
  
BOOL EnableSecurity();
```

Enable secure communication with the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **EnableSecurity** method enables a secure communications session with the remote host, automatically negotiating the encryption algorithm and validating the certificate specified by a previous call to the **SetCertificate** method. This method will cause the calling thread to block and wait for the client to initiate the TLS handshake.

This method is typically used to implement support for explicit TLS connections, where the client establishes a standard, non-secure connection to the server and then negotiates a secure connection at a later point. Usually this is done by the client sending a specific command to the server, and the server calls **EnableSecurity** from within the **OnRead** event handler that processes the command. If the method succeeds, all subsequent calls to **Read** and **Write** to receive and send data will be encrypted.

This method is only used to enable a secure connection for a specific client session. If all client connections should be secure, then call the **SetOptions** method to specify the `INET_OPTION_SECURE` option prior to starting the server and call the **SetCertificate** method to specify the server certificate that should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[DisableSecurity](#), [SetCertificate](#), [SetOptions](#)

CInternetServer::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the process level and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "0x". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxx:xxxx:xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[DisableTrace](#)

CInternetServer::EnumClients Method

```
INT EnumClients(  
    INTERNET_ADDRESS& ipAddress,  
    SOCKET *lpClients,  
    INT nMaxClients  
);  
  
INT EnumClients(  
    LPCTSTR lpszAddress,  
    SOCKET *lpClients,  
    INT nMaxClients  
);  
  
INT EnumClients(  
    SOCKET *lpClients,  
    INT nMaxClients  
);
```

Return a list of active client connections established with the server.

Parameters

ipAddress

A reference to an INTERNET_ADDRESS structure that contains the IP address that should be matched against the clients connected to the server. Only those clients that have connected to the server from this address will be returned in the *lpClients* array.

lpszAddress

A string that specifies the IP address that should be matched against the clients connected to the server. Only those clients that have connected to the server from this address will be returned in the *lpClients* array.

lpClients

Pointer to an array which will contain the socket handle for each active client session when the method returns. If this parameter is NULL, then the method will return the number of active client connections established with the server.

nMaxClients

Maximum number of socket handles to be returned in the *lpClients* array. If the *lpClients* parameter is NULL, this parameter should have a value of zero.

Return Value

If the method succeeds, the return value is the number of active client connections to the server. A return value of zero indicates that there are either no active client sessions, or no clients have connected using the specified IP address. If the method fails, the return value is FTP_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

If the *nMaxClients* parameter is less than the number of active client connections, the method will fail. To dynamically determine the number of active connections, call the method with the *lpClients* parameter with a value of NULL, and the *nMaxClients* parameter with a value of zero.

This method will not enumerate clients that have disconnected from the server, even if the session thread is still active. If the server is in the process of shutting down, this method will return zero,

indicating no active client sessions, even though there may be clients that are still in the process of disconnecting from the server. To determine the actual number of client sessions regardless of their status, use the **GetClientThreads** method.

Example

```
// Populate a listbox with the IP address for each client
pListBox->ResetContent();

INT nClients = pServer->EnumClients();
if (nClients > 0)
{
    SOCKET *phClients = new SOCKET[nClients];

    nClients = pServer->EnumClients(phClients, nClients);
    if (nClients == INET_ERROR)
    {
        // Unable to obtain list of connected clients
        return;
    }

    for (INT nIndex = 0; nIndex < nClients; nIndex++)
    {
        CString strAddress;

        if (pServer->GetClientAddress(phClients[nIndex], strAddress))
            pListBox->AddString(strAddress);
    }

    // Free the memory allocated for the socket handles
    delete phClients;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetClientAddress](#), [GetClientThreads](#)

CInternetServer::EnumNetworkAddresses Method

```
INT EnumNetworkAddresses(  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddressList,  
    INT nMaxAddresses  
);  
  
INT EnumNetworkAddresses(  
    LPINTERNET_ADDRESS lpAddressList,  
    INT nMaxAddresses  
);
```

The **EnumNetworkAddresses** method returns the list of network addresses that are configured for the local host.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return both IPv4 or IPv6 addresses for the local host, depending on how the system is configured and which network interfaces are enabled. This option is only recommended for applications that support IPv6.
INET_ADDRESS_IPV4	Specifies that the addresses should be in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the addresses should be in IPv6 format. All bytes in the <i>ipNumber</i> array are significant.

lpAddressList

A pointer to an array of [INTERNET_ADDRESS](#) structures that will contain the IP address of each local network interface.

nMaxAddresses

Maximum number of addresses to be returned.

Return Value

If the function succeeds, the return value is the number of network addresses that are configured for the local host. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the *nAddressFamily* parameter is specified as INET_ADDRESS_ANY, the application must be prepared to handle IPv6 addresses because it is possible that an IPv6 address string has been specified. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

If this method is called without specifying the address family, the value `INET_ADDRESS_IPV4` is used. This is provided primarily for compatibility with legacy applications and it is recommended that you explicitly specify the address family.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[FormatAddress](#), [GetHostAddress](#), [GetHostName](#)

CInternetServer::FindClient Method

```
SOCKET FindClient(  
    UINT nClientId  
);  
  
SOCKET FindClient(  
    LPCTSTR lpszMoniker  
);
```

Returns a handle to the client which matches the specified client ID or moniker.

Parameters

nClientId

An unsigned integer that specifies a unique client ID for the session. This value must be greater than zero.

lpszMoniker

A null-terminated string which specifies the client moniker to search for. This parameter cannot be NULL and cannot specify an empty string.

Return Value

If the method succeeds, the return value is the handle to the client socket for the session that matches the specified client ID or moniker. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call **GetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session using the **SetClientMoniker** method. This method will search all active client sessions for the server, and returns the socket handle to the client that matches the specified moniker. If there is no match, an error will be returned.

The moniker can be any string value, however monikers are not case sensitive and may not contain embedded null characters. The maximum length of a moniker is 127 characters.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetClientMoniker](#), [SetClientMoniker](#)

CInternetServer::Flush Method

```
BOOL Flush(  
    SOCKET hSocket  
);  
  
BOOL Flush();
```

Flush the send and receive buffers for the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Flush** method will flush any data waiting to be read or written to the remote host . It is important to note that this method is not similar to flushing data to a disk file; it does not ensure that a specific block of data has been written to the socket. For example, you should never call this function immediately after calling the **Write** method or prior to calling the **Disconnect** method.

An application never needs to use the **Flush** method under normal circumstances. This method is only to be used when the application needs to immediately return the socket to an inactive state with no pending data to be read or written. Calling this function may result in data loss and should only be used if you understand the implications of discarding any data which has been sent by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CInternetServer::FormatAddress Method

```
INT FormatAddress(  
    LPINTERNET_ADDRESS LpAddress,  
    LPTSTR LpszAddress,  
    INT cchAddress  
);  
  
INT FormatAddress(  
    LPINTERNET_ADDRESS LpAddress,  
    CString& strAddress  
);
```

The **FormatAddress** method converts a numeric IP address to a printable string. The format of the string depends on whether an IPv4 or IPv6 address is specified.

Parameters

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure which specifies the numeric IP address that should be converted to a string.

lpszAddress

The buffer that will contain the formatted IP address. This buffer should be at least 46 characters in length. This may also reference a **CString** object which will contain the formatted address when the method returns.

cchAddress

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is the length of the IP address string. If the method fails, the return value is `INET_ERROR`, meaning that the IP address could not be converted into a string. Typically this indicates that the pointer to the `INTERNET_ADDRESS` structure is invalid, or the data does not specify a valid IP address family.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetClientAddress](#), [GetLocalAddress](#), [INTERNET_ADDRESS](#)

InternetServer::GetActiveClient Method

SOCKET GetActiveClient();

Return the socket handle for the active client session.

Parameters

None.

Return Value

If the method succeeds, the return value is the socket handle for the active client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call the **GetLastError** method.

Remarks

The **GetActiveClient** method returns a handle to the client socket for the active client session. The active session is determined by the session thread that is currently executing, and therefore is only meaningful within the context of a server event handler such as **OnConnect** or **OnRead**. The value returned by this method is the same as the client socket handle that is passed to the event handler.

This method will fail within an **OnAccept** event handler because at that point the connection has not yet been accepted, therefore there is no active client session. It will also fail if called outside of an event handler. To obtain the socket handle associated with a particular session thread, use the **GetThreadClient** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSKV12.lib

See Also

[GetClientHandle](#), [GetClientThreadId](#), [GetThreadClient](#)

CInternetServer::GetAdapterAddress Method

```
INT GetAdapterAddress(  
    INT nAdapterIndex,  
    INT nAddressType,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);  
  
INT GetAdapterAddress(  
    INT nAdapterIndex,  
    INT nAddressType,  
    CString& strAddress  
);
```

Return the IP or MAC assigned to the specified network adapter.

Parameters

nAdapterIndex

An integer value that identifies the network adapter.

nAddressType

An integer value which specifies the type of address that should be returned:

Value	Description
INET_ADAPTER_IPV4	The address string will contain the primary IPv4 unicast address assigned to the network adapter.
INET_ADAPTER_IPV6	The address string will contain the primary IPv6 unicast address assigned to the network adapter.
INET_ADAPTER_MAC	The address string will contain the media access control (MAC) address assigned to the network adapter.

lpszAddress

A string buffer that will contain the IP or MAC address assigned to the adapter. This parameter cannot be NULL and it is recommended that it be at least 64 characters in length to provide enough space for any address type. An alternate form of the method accepts a **CString** argument which will contain the address.

nMaxLength

The maximum number of characters that can be copied into the string buffer, including the terminating null character. If the buffer is too small to store the complete address, this method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied to the string buffer, not including the terminating null character. A return value of zero indicates that the requested address type has not been assigned to the adapter. If the method fails, the return value is INET_ERROR and this typically indicates that either the adapter index is invalid or the string buffer is not large enough to store the complete address. To get extended error information, call **GetLastError**.

Remarks

The **GetAdapterAddress** method will return the IPv4, IPv6 or MAC address assigned to a specific network adapter. The primary network adapter has an index value of zero, and it increments for each adapter that is configured on the local system.

The media access control (MAC) address is a 48 bit or 64 bit value that is assigned to each network interface and is used for identification and access control. All network devices on the same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This method returns the MAC address string as sequence of hexadecimal values separated by a colon. An example of a 48 bit MAC address would be "01:23:45:67:89:AB". Note that some virtual network adapters may not have a MAC address assigned to them, in which case this method would return zero.

This method will ignore network adapters that have been disabled, as well as those that are bound to a virtual loopback interface. If the system has dial-up networking or virtualization software installed, this method may also return IP addresses assigned to a virtualized network adapters installed by that software.

Example

```
// Display the IPv4 address assigned to each network adapter
for (INT nIndex = 0;; nIndex++)
{
    CString strAddress;

    if (pServer->GetAdapterAddress(nIndex, INET_ADAPTER_IPV4, strAddress) ==
INET_ERROR)
        break;

    _tprintf(_T("Adapter %d: %s\n"), nIndex, szAddress);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[EnumNetworkAddresses](#), [GetLocalAddress](#), [GetLocalName](#)

CInternetServer::GetAddress Method

```
INT GetAddress(  
    LPCTSTR lpszAddress,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress  
);  
  
INT GetAddress(  
    LPCTSTR lpszAddress,  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **GetAddress** method converts an IP address string to binary format.

Parameters

lpszAddress

A null-terminated string which specifies an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address specified by the *lpszAddress* parameter. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the value of the <i>lpszAddress</i> parameter.
INET_ADDRESS_IPV4	Specifies that the address should be in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero. If the <i>lpszAddress</i> parameter does not specify a valid IPv4 address string, this method will fail.
INET_ADDRESS_IPV6	Specifies that the address should be in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. If the <i>lpszAddress</i> parameter does not specify a valid IPv6 address string, this method will fail.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the *nAddressFamily* parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible that an IPv6 address string has been specified. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member

should always be specified as `INET_ADDRESS_IPV4` to ensure that only IPv4 addresses are returned and any attempt to specify an IPv6 address string would result in an error.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpzAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

Unicode: Implemented as Unicode and ANSI versions

See Also

[FormatAddress](#), [IsAddressNull](#), [IsAddressRoutable](#), [IsProtocolAvailable](#), [INTERNET_ADDRESS](#)

CInternetServer::GetAddressFamily Method

```
INT GetAddressFamily(  
    LPCTSTR lpszAddress,  
);
```

Return the address family for the specified IP address.

Parameters

lpszAddress

A null-terminated string which specifies an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

Return Value

If the method succeeds, the return value is the address family for the specified IP address and may be one of the values listed below. If the method fails, the return value is INET_ADDRESS_UNKNOWN. To get extended error information, call the **GetLastError** method.

Value	Description
INET_ADDRESS_IPV4	The address passed to the method is a valid IPv4 address.
INET_ADDRESS_IPV6	The address passed to the method is a valid IPv6 address.

Remarks

The **GetAddressFamily** method returns the address family associated with the specified IP address string. This can be used to determine if a string specifies a valid IPv4 or IPv6 address that can be passed to other methods such as **Connect**. Note that this method will not attempt to resolve hostnames, it will only accept IP addresses.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpszAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

- Minimum Desktop Platform:** Windows 7 Service Pack 1
- Minimum Server Platform:** Windows Server 2008 R2 Service Pack 1
- Header File:** cswsock12.h
- Import Library:** cswskv12.lib
- Unicode:** Implemented as Unicode and ANSI versions

See Also

[IsAddressNull](#), [IsAddressRoutable](#), [IsProtocolAvailable](#), [INTERNET_ADDRESS](#)

CInternetServer::GetBacklog Method

```
UINT GetBacklog();
```

Return the size of the backlog connection queue for the server.

Parameters

None.

Return Value

The return value is the size of the queue used to accept client connections.

Remarks

The **GetBacklog** method returns the size of the queue allocated for pending client connections. A value of zero specifies that the size of the queue should be set to a reasonable default value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections. To change the size of the backlog queue, use the **SetBacklog** method prior to starting the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cswskv12.lib

See Also

[SetBacklog](#), [Start](#), [Data Members](#)

CInternetServer::GetClientAddress Method

```
INT GetClientAddress(  
    SOCKET hSocket,  
    LPINTERNET_ADDRESS LpAddress,  
    UINT *LpnRemotePort  
);  
  
INT GetClientAddress(  
    SOCKET hSocket,  
    LPTSTR LpszAddress,  
    INT nMaxLength  
);  
  
INT GetClientAddress(  
    SOCKET hSocket,  
    CString& strAddress  
);
```

Return the IP address and port number for the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to a server or client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the client that is connected to the server. This parameter may be NULL, in which case the IP address will not be returned to the caller.

lpnRemotePort

A pointer to an unsigned integer that will contain the port number of the client that is connected to the server. This parameter may be NULL, in which case the port number will not be returned to the caller.

lpszAddress

A null-terminated string buffer that will contain the formatted IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length.

nMaxLength

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

If this method is called within an **OnAccept** event handler, passing the server handle as the *hSocket* parameter will return the IP address and port number for the client that is attempting to establish the connection. If the client address is unavailable, the *ipFamily* member of the `INTERNET_ADDRESS` structure will be zero.

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetClientPort](#), [OnAccept](#), [OnConnect](#), [INTERNET_ADDRESS](#)

CInternetServer::GetClientData Method

```
BOOL GetClientData(  
    SOCKET hClient,  
    LPVOID *lppvData  
);
```

```
BOOL GetClientData(  
    LPVOID *lppvData  
);
```

Returns the application defined data associated with the specified client session.

Parameters

hClient

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lppvData

A pointer to a void pointer which will contain an application defined value associated with the client session.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the client session could not be retrieved. To get extended error information, call **GetLastError**.

Remarks

The **GetClientData** method is used to retrieve the application defined data that was previously associated with a client session using the **SetClientData** method. This is typically used to associate a pointer to a data structure or a class instance with a specific client handle.

This method can only be used with client socket handles created using the server interface. If the socket handle is invalid, or does not reference a client socket handle created by the server, the *lppvData* pointer passed to this method will be initialized to a value of NULL and the method will return a value of zero.

If this method is called with a valid socket handle and there is no data associated with the socket, the method will return a non-zero value and the *lppvData* pointer will be returned with a NULL value. Before dereferencing the pointer returned by this method, the application should always check the return value to ensure the method succeeded and make sure that the pointer is not NULL.

Example

```
UINT *pnValue1 = new UINT;  
UINT *pnValue2 = NULL;  
  
*pnValue1 = 1234;  
  
if (!pServer->SetClientData(hClient, pnValue1))  
{  
    // Unable to associate the data with this session  
    return;  
}
```

```
if (!pServer->GetClientData(hClient, (LPVOID *)&pnValue2))
{
    // Unable to retrieve the data associated with this session
    return;
}

// pnValue2 == pnValue1
printf("The value of the user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[SetClientData](#)

CIInternetServer::GetClientHandle Method

```
SOCKET GetClientHandle(  
    UINT nClientId  
);
```

Returns the handle for a specific client session based on its ID number.

Parameters

nClientId

An unsigned integer value which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is the socket handle for the specified client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call **GetLastError**.

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value called the client ID. The **GetClientHandle** method will return the socket handle that is associated with the specified client ID. Unlike socket handles, which are reused by the operating system, the client ID is guaranteed to be unique throughout the lifetime of the server. To obtain the ID associated with the client session, use the **GetClientId** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetClientId](#), [GetClientMoniker](#), [SetClientMoniker](#), [GetThreadClient](#)

CIInternetServer::GetClientId Method

```
UINT GetClientId(  
    SOCKET hSocket  
);  
  
UINT GetClientId();
```

Returns the unique ID number assigned to the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is an unsigned integer value which uniquely identifies the client session. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value. This value can be obtained by calling the **GetClientId** method and used by the application to identify that client session. The **GetClientHandle** method can then be used to obtain the client socket handle for the session, based on that client ID. It is important to note that the actual value of the client ID should be considered opaque. It is only guaranteed that the value will be greater than zero, and that it will be unique to the client session.

While it is possible for a client socket handle to be reused by the operating system, client IDs are unique throughout the life of the server session and are never duplicated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetClientHandle](#), [GetClientMoniker](#), [SetClientMoniker](#)

CIInternetServer::GetClientIdleTime Method

```
DWORD GetClientIdleTime(  
    SOCKET hSocket  
);  
  
DWORD GetClientIdleTime();
```

Returns the number of milliseconds that the specified client session has been idle.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is an unsigned integer value which specifies the number of milliseconds the client session has been idle. If the method fails, the return value is INFINITE. To get extended error information, call **GetLastError**.

Remarks

The **GetClientIdleTime** method will return the number of milliseconds that have elapsed since data was exchanged with the client. The elapsed time is limited to the resolution of the system timer, which is typically in the range of 10 milliseconds to 16 milliseconds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetClientHandle](#), [GetClientId](#), [GetClientMoniker](#), [GetTimeout](#)

CInternetServer::GetClientMoniker Method

```
INT GetClientMoniker(  
    SOCKET hClient,  
    LPTSTR lpszMoniker,  
    INT nMaxLength  
);  
  
INT GetClientMoniker(  
    SOCKET hClient,  
    CString& strMoniker  
);
```

The **GetClientMoniker** method returns the moniker associated with the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpszMoniker

Pointer to a string buffer that will contain the moniker for the specified client session when the method returns. An alternate version of this method accepts a **CString** object if it is available.

nMaxLength

The maximum number of characters that may be copied into the string buffer. The buffer must be large enough to store the moniker and a terminating null character. The maximum length of a moniker is 127 characters.

Return Value

If the method succeeds, the return value is the number of characters in the moniker string. A return value of zero specifies that no moniker was assigned to the socket. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session using the **SetClientMoniker** method. This method will return the moniker that was previously assigned to the client, if any. To obtain the socket handle associated with a given moniker, use the **FindClient** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[FindClient](#), [GetClientId](#), [SetClientMoniker](#)

CInternetServer::GetClientPort Method

```
INT GetClientPort(  
    SOCKET hSocket  
);  
  
INT GetClientPort();
```

Returns the remote port number used by the client to establish the connection.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is the port number. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **GetClientPort** method returns the remote port number that the client is bound to. Note that this is not the port number that the server is using to listen for connections, it is the port number that the client is bound to on the remote host. Typically this is an ephemeral port, either in the range of 1025 through 5000, or greater than 32768, depending on the client operating system.

If this method is called within the **OnAccept** event handler, providing the server socket handle as the *hSocket* parameter will return the port number of the client that is attempting to establish the connection.

It is not recommended that you use the client port number for anything other than informational and logging purposes. Do not make any assumptions about the specific port number or range of port numbers that a client is using when establishing a connection to the server. The ephemeral port number that a client is bound to can vary based on the client operating system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[GetClientAddress](#), [OnAccept](#), [OnConnect](#)

InternetServer::GetClientServer Method

```
SOCKET HttpGetClientServer(  
    UINT nClientId  
);
```

The **GetClientServer** method returns a handle to the server that created the specified client session.

Parameters

nClientId

An unsigned integer which uniquely identifies the client session.

Return Value

If the method succeeds, the return value is the handle to the server that created the client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientServer** method returns the handle to the server that created the client session and is typically used within a notification message handler. If the server is in the process of shutting down, or the client session thread is terminating, this method will fail and return INVALID_SOCKET indicating that the session ID is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[AsyncNotify](#)

CInternetServer::GetClientServerById Method

```
SOCKET GetClientServerById(  
    UINT nClientId  
);
```

The **GetClientServerById** method returns a socket handle to the server for the specified client session identifier.

Parameters

nClientId

Client session identifier.

Return Value

If the method succeeds, the return value is the handle to the server that created the client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientServerById** method returns the handle to the server that created the client session using the client's unique identifier. The **GetClientServer** method can be used to obtain the server handle using the client socket handle rather than the client session ID. This method is typically used in conjunction with the INET_NOTIFY_CONNECT notification message to obtain the handle to the server that generated the event using the client ID passed in the *wParam* message parameter.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[AsyncNotify](#), [GetClientHandle](#), [GetClientId](#), [GetClientServer](#)

InternetServer::GetClientThreadId Method

```
DWORD GetClientThreadId(  
    SOCKET hSocket  
);  
  
DWORD GetClientThreadId();
```

Returns the thread ID associated with the specified client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

Return Value

If the method succeeds, the return value is a thread ID. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **GetClientThreadId** method returns a thread ID that can be used to identify the thread that is managing the client session. The thread ID can be used with other Windows API functions such as **OpenThread**. Exercise caution when using thread-related functions, interfering with the normal operation of the thread can have unexpected results. You should never use this method to obtain a thread handle and then call the **TerminateThread** function to terminate a client session. This will prevent the thread from releasing the resources that were allocated for the session and can leave the server in an unstable state. To terminate a client session, use the **Disconnect** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

See Also

[EnumClients](#), [GetActiveClient](#)

InternetServer::GetClientThreads Method

```
INT GetClientThreads();
```

Returns the number of client session threads created by the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of client session threads that have been created by the server. If the method fails, the return value is `INET_ERROR`. To get extended error information, call the **GetLastError** method.

Remarks

The **InetGetClientThreads** function returns the number of threads that are managing client sessions for the specified server. If there are no clients connected to the server, this function will return a value of zero. Because this function returns the number of session threads, the value returned will include those clients that are in the process of disconnecting from the server but their session thread has not yet terminated. This differs from the **InetEnumServerClients** function which will only enumerate active clients.

If you wish to determine when the last client has disconnected from the server, call this function within an event handler for the `INET_EVENT_DISCONNECT` event. If the function returns a value greater than one, then there are other client sessions that are either connected or in the process of terminating.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[EnumClients](#)

CIInternetServer::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT nMaxLength  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CInternetServer::GetExternalAddress Method

```
INT GetExternalAddress(  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress,  
    INT nMaxLength  
);  
  
INT GetExternalAddress(  
    INT nAddressFamily,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);  
  
INT GetExternalAddress(  
    INT nAddressFamily,  
    CString& strAddress  
);
```

The **GetExternalAddress** method returns the external IP address for the local system.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Value	Description
INET_ADDRESS_IPV4	Specifies that the address should be in IPv4 format. The method will attempt to determine the external IP address using an IPv4 network connection.
INET_ADDRESS_IPV6	Specifies that the address should be in IPv6 format. The method will attempt to determine the external IP address using an IPv6 network connection and requires that the local host have an IPv6 network interface installed and enabled.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the external IP address of the local host in binary form.

lpszAddress

A null-terminated string buffer that will contain the external IP address of the local host.

nMaxLength

The maximum length of the string that will contain the IP address when the method returns.

Return Value

In the first form of the method, if it succeeds, the return value is the IP address of the local system in numeric form. If the method fails, the return value is INET_ADDRESS_NONE. In the second form, the return value is the length of the IP address string and an error is indicated by the return value INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **GetExternalAddress** method returns the IP address assigned to the router that connects the local host to the Internet. This is typically used by an application executing on a system in a local network that uses a router which performs Network Address Translation (NAT). In that network configuration, the **GetLocalAddress** method will only return the IP address for the local system on the LAN side of the network unless a connection has already been established to a remote host. The **GetExternalAddress** function can be used to determine the IP address assigned to the router on the Internet side of the connection and can be particularly useful for servers running on a system behind a NAT router.

This method requires that you have an active connection to the Internet and calling this function on a system that uses dial-up networking may cause the operating system to automatically connect to the Internet service provider. An application should always check the return value in case there is an error; never assume that the return value is always a valid address. The function may be unable to determine the external IP address for the local host for a number of reasons, particularly if the system is behind a firewall or uses a proxy server that restricts access to external sites on the Internet. If the function is able to obtain a valid external address for the local host, that address will be cached by the library for sixty minutes. Because dial-up connections typically have different IP addresses assigned to them each time the system is connected to the Internet, it is recommended that this function only be used with broadband connections where a NAT router is being used.

Calling this function may cause the current thread to block until the external IP address can be resolved and should never be used in conjunction with asynchronous socket connections. If you need to call this function in an application which uses asynchronous sockets, it is recommended that you create a new thread and call this function from within that thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetClientAddress](#), [GetHostAddress](#), [GetLocalAddress](#)

CIInternetServer::GetHandle Method

SOCKET GetHandle();

The **GetHandle** method returns the socket handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the socket handle associated with the current instance of the class object. If there is no active connection, the value INVALID_SOCKET will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketWrench API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CInternetServer::GetHostAddress Method

```
INT GetHostAddress(  
    LPCTSTR lpszHostName,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS LpAddress  
);  
  
INT GetHostAddress(  
    LPCTSTR lpszHostName,  
    LPINTERNET_ADDRESS LpAddress  
);
```

The **InetGetHostAddress** method resolves the specified host name into an IP address in binary format.

Parameters

lpszHostName

The name of the host to resolve; this may be a fully-qualified domain name or an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on how the host name can be resolved. By default, a preference will be given for returning an IPv4 address. However, if the host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the specified host.

Return Value

If the method succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

This method can also be used to convert an address in dot notation to a binary format. If the method must perform a DNS lookup to resolve the hostname, the calling thread will block. To ensure future

compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the *ipFamily* member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

The *nAddressFamily* parameter is used to specify a preference for the type of address returned, however it is possible that a host may not have an IPv4 or IPv6 address record, in which case this function will fail. Although IPv4 is still the most common address used at this time, an application should not assume that because a given host name does not have an IPv4 address, that the host name is invalid.

If the *nAddressFamily* parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible for a host name to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpzHostName* parameter specifies an host that only has an IPv6 (AAAA) DNS record.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetClientAddress](#), [GetHostName](#), [GetLocalAddress](#), [GetLocalName](#), [IsProtocolAvailable](#), [INTERNET_ADDRESS](#)

CInternetServer::GetHostName Method

```
INT GetHostName(  
    LPINTERNET_ADDRESS LpAddress,  
    LPTSTR LpszHostName,  
    INT cchHostName  
);  
  
INT GetHostName(  
    LPINTERNET_ADDRESS LpAddress,  
    CString& strHostName  
);
```

The **GetHostName** method performs a reverse lookup, returning the host name associated with a given IP address.

Parameters

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure which specifies the IP address that should be resolved into a host name.

lpszHostName

The buffer that will contain the host name. It is recommended that this buffer be at least 256 characters in length to accommodate the longest possible fully qualified domain name. This parameter cannot be NULL. An alternate form of the method accepts a **CString** argument which will contain the hostname.

cchHostName

The maximum number of characters that can be copied into the buffer.

Return Value

If the method succeeds, the return value is the length of the hostname. If the method fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the method must perform a reverse DNS lookup to resolve the IP address into a host name, the calling thread will block. This method requires that the host have a PTR record, otherwise it will fail. Because many hosts do not have a PTR record, calling this method frequently may have a negative impact on the overall performance of the application.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetClientAddress](#), [GetHostAddress](#), [GetLocalAddress](#), [GetLocalName](#), [IsProtocolAvailable](#),

CInternetServer::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CInternetServer::GetLocalAddress Method

```
INT GetLocalAddress(  
    LPINTERNET_ADDRESS LpAddress,  
    UINT *LpnPort  
);  
  
INT GetLocalAddress(  
    LPTSTR LpszAddress,  
    INT nMaxLength  
);  
  
INT GetLocalAddress(  
    CString& strAddress  
    UINT *LpnPort  
);
```

Return the local IP address and port number for the server.

Parameters

LpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the local host. If the server is not active, this function will attempt to determine the IP address of the local host assigned by the system. If the address is not required, this parameter may be NULL.

LpszAddress

A null-terminated string that will contain the IP address of the local host. If this version of the method is used, the IP address is converted to a string format using the **FormatAddress** method. The string should be able to store at least 46 characters to ensure that both IPv4 and IPv6 formatted addresses can be returned without the possibility of a buffer overrun. An alternate form of the method accepts a **CString** argument which will contain the line of text returned by the server.

LpnPort

A pointer to an unsigned integer that will contain the local port number. If the server is active, this parameter will be set to the local port that the listening socket was bound to. If the server is not active, this parameter is ignored. If the port number is not required, this parameter may be NULL.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

To ensure future compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the *ipFamily* member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

If the system is connected to the Internet through a local network and/or uses a router that performs Network Address Translation (NAT), the **GetLocalAddress** method will return the local, non-routable IP address assigned to the local system. To determine the public IP address has been assigned to the system, you should use the **GetExternalAddress** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cssock12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetClientAddress](#), [GetExternalAddress](#), [GetHostAddress](#), [GetHostName](#), [GetLocalName](#),
[INTERNET_ADDRESS](#)

InternetServer::GetLocalName Method

```
INT GetLocalName(  
    LPTSTR lpszHostName,  
    INT cchHostName  
);  
  
INT GetLocalName(  
    CString& strHostName  
);
```

The **GetLocalName** method returns the hostname assigned to the local system.

Parameters

lpszHostName

The buffer that will contain the hostname. This parameter cannot be NULL. An alternate form of the method accepts a **CString** argument which will contain the local hostname.

cchHostName

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is the length of the hostname. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetClientAddress](#), [GetLocalAddress](#)

CInternetServer::GetMemoryUsage Method

```
SIZE_T GetMemoryUsage();
```

Return the amount of memory allocated for the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of bytes allocated. If the method fails, the return value is zero. Call the **GetLastError** method to determine the cause of the failure.

Remarks

This method returns the amount of memory allocated for the server. It enumerates all heap allocations made by the process and returns the total number of bytes allocated for the server and all client sessions. This value only reflects the amount of memory explicitly allocated by this class library and does not reflect the total working set size of the process, or memory allocated by any other libraries. To determine the working set size for the process, refer to the Win32 **GetProcessWorkingSetSize** and **GetProcessMemoryInfo** functions.

This method forces the server into a locked state, and all network operations will block until the method returns. Because this method enumerates all heaps created for the process, it can be an expensive operation, particularly when a large number of connections have been established. Frequent use of this method can significantly degrade the performance of the application. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active client sessions. If the value returned by the method remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

CInternetServer::GetOptions Method

DWORD GetOptions();

Return the current server options.

Parameters

None.

Return Value

This method returns an unsigned integer value that specifies the server options that are currently enabled for the class instance. For a list of the available options, see [Server Option Constants](#). This method returns the value of the **m_dwOptions** class member variable.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[SetOptions](#), [Data Members](#)

CInternetServer::GetPriority Method

INT GetPriority();

Return the current priority assigned to the specified server.

Parameters

None.

Return Value

If the method succeeds, the return value is the priority for the specified server. If the method fails, the return value is INET_PRIORITY_INVALID. To get extended error information, call the **GetLastError** method.

Remarks

The **GetPriority** method can be used to determine the current priority assigned to the server. It will return one of the following values:

Value	Description
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

See Also

[SetPriority](#)

CInternetServer::GetStackSize Method

DWORD GetStackSize();

Return the initial size of the stack allocated for threads created by the server.

Parameters

None.

Return Value

If the method succeeds, the return value is the amount of memory that will be allocated for the stack in bytes. If the method fails, the return value is zero. To get extended error information, call

GetLastError.

Remarks

The **GetStackSize** method returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[SetStackSize](#), [Start](#)

CInternetServer::GetStatus Method

```
INT GetStatus();
```

Return the current status of the server.

Parameters

None.

Return Value

An integer value that specifies the server status.

Remarks

The return value is one of the following values:

Value	Description
INET_SERVER_INACTIVE	The server is currently inactive.
INET_SERVER_STARTED	The server has initialized and is preparing to listen for client connections.
INET_SERVER_LISTENING	The server is actively listening for incoming client connections.
INET_SERVER_SUSPENDED	The server has been suspended and is no longer accepting client connections.
INET_SERVER_SHUTDOWN	The server has been stopped and is in the process of terminating all active client connections.

Requirements

- Minimum Desktop Platform:** Windows 7 Service Pack 1
- Minimum Server Platform:** Windows Server 2008 R2 Service Pack 1
- Header File:** cswsock12.h
- Import Library:** cswskv12.lib

See Also

[IsInitialized](#), [IsListening](#), [IsLocked](#)

CIInternetServer::GetStreamInfo Method

```
BOOL GetStreamInfo(  
    SOCKET hSocket  
    LPINETSTREAMINFO lpStreamInfo  
);  
  
BOOL GetStreamInfo(  
    LPINETSTREAMINFO lpStreamInfo  
);
```

The **GetStreamInfo** function fills a structure with information about the current stream I/O operation.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpSecurityInfo

A pointer to an **INETSTREAMINFO** structure which contains information about the status of the current operation.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetStreamInfo** method returns information about the current streaming socket operation, including the average number of bytes transferred per second and the estimated amount of time until the operation completes. If there is no operation currently in progress, this method will return the status of the last successful streaming read or write performed by the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSKV12.lib

See Also

[ReadStream](#), [StoreStream](#), [WriteStream](#), [INETSTREAMINFO](#)

InternetServer::GetThreadClient Method

```
SOCKET WINAPI InetGetThreadClient(  
    DWORD dwThreadId  
);
```

The **GetThreadClient** method returns the socket handle for the client session that is being managed by the specified thread.

Parameters

dwThreadId

An unsigned integer value which identifies the thread managing the client session. If this parameter has a value of zero, then the client handle for the current thread is returned.

Return Value

If the method succeeds, the return value is the socket handle for the specified client session. If the method fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The **GetThreadClient** is used to obtain the socket handle for the client session that is being managed by the specified thread. If the specified thread ID is zero, then the method will return the client socket for the current thread, otherwise it will search the internal table of all active client sessions and return the handle to the session that is being managed by that thread.

This method will fail if the thread ID does not specify an active client session thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetActiveClient](#), [GetClientHandle](#), [GetClientId](#), [GetClientThreadId](#)

InternetServer::GetTimeout Method

```
INT GetTimeout();
```

Return the timeout interval for blocking network operations in seconds.

Parameters

None.

Return Value

The return value is the timeout period in seconds. If there is no active server, this will return the timeout period that will be used when the server is started. A value of zero specifies that a reasonable default timeout period will be automatically selected.

Remarks

The **GetTimeout** method returns the number of seconds the server will wait for a blocking network operation to complete, such as sending or receiving data. This value also determines the amount of time that the server will wait for the client to send data before invoking the **OnTimeout** event handler for that session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[IsReadable](#), [IsWritable](#), [SetTimeout](#), [OnTimeout](#)

CInternetServer::IsActive Method

BOOL IsActive();

Determine if the server has been started.

Return Value

This method returns a non-zero value if the server has been started. If the server is stopped this method will return zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[CInternetServer](#), [IsListening](#), [Start](#), [Stop](#)

CInternetServer::IsAddressNull Method

```
BOOL IsAddressNull(  
    LPCTSTR lpszAddress  
);  
  
BOOL IsAddressNull(  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **IsAddressNull** method determines if the IP address is null.

Parameters

lpszAddress

A string that specifies the IP address.

lpAddress

A pointer to an INTERNET_ADDRESS structure that specifies the IP address.

Return Value

If the method succeeds and the IP address is null, or the parameter is a NULL pointer, the return value is non-zero. If the method fails or the address is not null, the return value is zero. If the address family is not supported, the last error code will be updated. If the address is valid but not null, the last error code will be set to NO_ERROR.

Remarks

A null IP address is one where all bits for the address (32 bits for IPv4 or 128 bits for IPv6) are zero. This is a special address that is typically used when creating a passive socket that should listen for connections on all available network interfaces.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetAddress](#), [IsAddressRoutable](#), [INTERNET_ADDRESS](#)

CInternetServer::IsAddressRoutable Method

```
BOOL IsAddressRoutable(  
    LPCTSTR lpszAddress  
);  
  
BOOL IsAddressRoutable(  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **IsAddressRoutable** method determines if the IP address is routable over the Internet.

Parameters

lpszAddress

A string that specifies the IP address.

lpAddress

A pointer to an INTERNET_ADDRESS structure that specifies the IP address.

Return Value

If the method succeeds and the IP address is routable over the Internet, the return value is non-zero. If the method fails or the address is not routable, the return value is zero. If the parameter is NULL, or the address family is not supported, the last error code will be updated. If the address is valid but not routable, the last error code will be set to NO_ERROR.

Remarks

A routable IP address is one that can be reached by anyone over the public Internet. These are also commonly referred to as "public addresses" which are typically assigned to networks and individual hosts by an Internet service provider. There are also certain addresses that are not routable over the Internet, and used to address systems over a local network or private intranet. This function can be used to determine if a given IP address is public (routable) or private.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetAddress](#), [GetExternalAddress](#), [IsAddressNull](#), [INTERNET_ADDRESS](#)

CInternetServer::IsInitialized Method

BOOL IsInitialized();

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[CInternetServer](#), [IsListening](#), [IsLocked](#)

InternetServer::IsListening Method

BOOL IsListening();

Determine if the server is listening for client connections.

Parameters

None.

Return Value

If the server has started and is listening for client connections, the method returns a non-zero value. If the server is not listening for connections, the return value is zero.

Remarks

The **IsListening** method determines if the server has been started and is actively listening for incoming connection requests from client applications. This method will return zero if the server is not active, if it has been suspended using the **Suspend** method or if the **Stop** method has been called to shutdown the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[IsActive](#), [Start](#), [Stop](#), [Suspend](#)

CInternetServer::IsLocked Method

BOOL IsLocked();

Determine if the server is currently in a locked state.

Parameters

None.

Return Value

If the server is locked, the method returns a non-zero value. If the server is not locked, the return value is zero.

Remarks

The **IsLocked** method determines if a server has been locked using the **Lock** method. Only the thread that has locked the server may interact with it and all other threads will block when they attempt to perform a network operation. After the server is unlocked, the blocked threads will resume normal execution. If the application has created multiple instances of the **CInternetServer** class, this method will return a non-zero value if any of those servers have been locked, not only the current instance.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[Lock](#), [Unlock](#)

CInternetServer::IsProtocolAvailable Method

```
BOOL IsProtocolAvailable(  
    INT nAddressFamily,  
    INT nProtocol  
);
```

The **IsProtocolAvailable** method determines if the operating system supports creating a socket for the specified address family and protocol.

Parameters

nAddressFamily

An integer which identifies the address family that should be checked. It should be one of the following values:

Value	Description
INET_ADDRESS_IPV4	Specifies that the function should determine if it can create an Internet Protocol version 4 (IPv4) socket. This requires that the system have an IPv4 TCP/IP stack bound to at least one network adapter on the local system. All Windows systems include support for IPv4 by default.
INET_ADDRESS_IPV6	Specifies that the function should determine if it can create an Internet Protocol version 6 (IPv6) socket. This requires that the system have an IPv6 TCP/IP stack bound to at least one network adapter on the local system. Windows XP and Windows Server 2003 includes support for IPv6, however it is not installed by default. Windows Vista and later versions include support for IPv6 and enable it by default.

nProtocol

An integer which identifies the protocol that should be checked. It should be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This requires that the system be capable of creating a stream socket using the specified address family.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. This requires that the system be capable of creating a datagram socket using the specified address family.

Return Value

If the system is capable of creating a socket using the specified address family and protocol, this method will return a non-zero value. If the combination of address family and protocol is not supported, this method will return a value of zero.

Remarks

The **IsProtocolAvailable** method is used to determine if the operating system supports creating a particular type of socket. Typically it is used by an application to determine if the system has an IPv6 TCP/IP stack installed and configured. By default, all Windows systems will have an IPv4 stack installed if the system has a network adapter. However, not all systems may have an IPv6 stack installed, particularly older Windows XP and Windows Server 2003 systems. Note that if an IPv6 stack is not installed, the library will not recognize IPv6 addresses and cannot resolve host names that only have an IPv6 (AAAA) record, even if the address or host name is valid.

Example

```
if (!pSocket->IsProtocolAvailable(INET_ADDRESS_IPV6, INET_PROTOCOL_TCP))
{
    AfxMessageBox(_T("This system does not support IPv6"), MB_ICONEXCLAMATION);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetAddress](#), [GetHostAddress](#), [GetHostName](#)

CInternetServer::IsReadable Method

```
BOOL IsReadable();  
  
BOOL IsReadable(  
    SOCKET hSocket  
);  
  
BOOL IsReadable(  
    SOCKET hSocket,  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **IsReadable** method is used to determine if data is available to be read from the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

nTimeout

Timeout for remote host response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the server can read data from the client within the specified timeout period, the method returns a non-zero value. If there is no data available to be read, the method returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

If the connection is secure, the value returned in *lpdwAvail* will reflect the number of bytes available in the encrypted data stream. The actual amount of data available to the application after it has been decrypted will vary.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[IsWritable](#), [Peek](#), [Read](#), [ReadLine](#), [ReadStream](#)

CInternetServer::IsWritable Method

```
BOOL IsWritable(  
    INT nTimeout  
);
```

The **IsWritable** method is used to determine if data can be written to the remote host.

Parameters

nTimeout

Timeout for remote host response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If data can be sent to the client within the specified timeout period, the method returns a non-zero value. The method will return zero if the socket send buffer is full.

Remarks

The **IsWritable** method cannot be used to determine the amount of data that can be sent to the client without blocking the current thread. A non-zero return value only indicates that the send buffer is not full and can accept some data. In most cases, it is recommended that larger blocks of data be broken into smaller logical blocks rather than attempting to send it all of the data at once. For very large streams of data, it is recommended that you use the **WriteStream** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[IsReadable](#), [Write](#), [WriteLine](#), [WriteStream](#)

CInternetServer::Lock Method

BOOL Lock();

Lock the server, causing other client threads to block until it is unlocked.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **Lock** method causes the specified server to enter a locked state where only the current thread may interact with the server and the clients that are connected to it. While a server is locked, all other threads will block when they attempt to perform a network operation. When the server is unlocked, the blocked threads will resume normal execution.

This method should be used carefully, and a server should never be left in a locked state for an extended period of time. It is meant to be used when the server process updates a global data structure and it must prevent any other threads from performing a network operation during the update. Only one server can be locked at any one time, and once a server has been locked, it can only be unlocked by the same thread.

The program should always check the return value from this method, and should never assume that the lock has been established. If more than one thread attempts to lock a server at the same time, there is no guarantee as to which thread will actually establish the lock. If a potential deadlock situation is detected, this method will fail and return a value of zero.

Every time the **Lock** method is called, an internal lock counter is incremented, and the lock will not be released until the lock count drops to zero. This means that each call to the **Lock** method must be matched by an equal number of calls to the **Unlock** method. Failure to do so will result in the server becoming non-responsive as it remains in a locked state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

See Also

[IsLocked](#), [Unlock](#)

CInternetServer::MatchHostName Method

```
BOOL MatchHostName(  
    LPCTSTR lpszHostName,  
    LPCTSTR lpszHostMask  
    BOOL bResolve  
);
```

The **MatchHostName** method matches a host name against one or more strings that may contain wildcards.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address to match.

lpszHostMask

A null-terminated string which specifies one or more values to match against the host name. The asterisk character can be used to match any number of characters in the host name, and the question mark can be used to match any single character. Multiple values may be specified by separating them with a semicolon.

bResolve

A boolean value which specifies if the host name or IP address should be resolved when matching the host against the mask string. If this parameter is non-zero, two checks against the host mask string will be performed; once for the host name specified and once for its IP address. If this parameter is zero, then the match is made only against the host name string provided.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **MatchHostName** method provides a convenient way for an application to determine if a given host name matches one or more mask strings which may contain wildcard characters. For example, the host name could be "www.microsoft.com" and the host mask string could be "*.microsoft.com". In this example, the method would return a non-zero value indicating the host name matched the mask. However, if the mask string was "*.net" then the method would return zero, indicating that there was no match. Multiple mask values can be combined by separating them with a semicolon; for example, the mask "*.com;*.org" would match any host name in either the .com or .org top-level domains.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetAddress](#), [GetHostAddress](#), [GetHostName](#)

CInternetServer::Peek Method

```
INT Peek(  
    SOCKET hSocket,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Peek(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

Read data from the client without removing it from the socket buffer.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpBuffer

Pointer to the buffer in which the data will be stored. This argument may be NULL, in which case no data is copied from the socket buffers, however the function will return the number of bytes available to read.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. If the *lpBuffer* parameter is not NULL, this value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes available to read from the socket. A return value of zero indicates that there is no data available to read at that time. If the function fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **Peek** method returns data that is available to read from the socket, up to the number of bytes specified. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation, up to the specified buffer size. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. To determine if there is data available to be read, use the **IsReadable** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[IsReadable](#), [IsWritable](#), [Read](#), [ReadLine](#), [Write](#), [WriteLine](#)

CInternetServer::PreProcessEvent Method

```
virtual LONG PreProcessEvent(  
    SOCKET hServer,  
    UINT nClientId,  
    UINT nEventId,  
    DWORD dwError,  
    BOOL& bHandled  
);
```

A virtual method that is invoked for each event generated by the server.

Parameters

hServer

The server handle. The application should treat this as an opaque value that is only valid as long as the server is active. This value should not be stored by the application and the handle value will change if the server is restarted.

nClientId

An unsigned integer which uniquely identifies the client that has issued a request to the server. This value is guaranteed to be unique to the client session throughout the life of the server and is never reused. The application should never make assumptions about the order in which IDs are allocated to the client sessions.

nEventId

An unsigned integer which specifies which event occurred. For a list of events, see [Server Event Constants](#).

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

bHandled

An integer which specifies if the event has been handled by the application. If this parameter is set to a non-zero value, the default event handler will not be invoked for the event.

Return Value

The method should return a value of zero to indicate that the default event handler should be invoked for the event. If the method returns a non-zero value, this value is passed back to the event dispatcher and the default handler will not be invoked.

Remarks

The **PreProcessEvent** method is invoked for each event that is generated, prior to the default handler for that event. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

CInternetServer::Read Method

```
INT Read(  
    SOCKET hSocket,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Read(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

Read data from the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpBuffer

Pointer to the buffer in which the data will be stored. This parameter cannot be NULL.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes read. A return value of zero indicates that the client has closed the connection and there is no more data available to be read. If the method fails, the return value is `INET_ERROR`. To get extended error information, call the **GetLastError** method.

Remarks

The **Read** method will read up to the specified number of bytes and store the data in the buffer provided by the caller. If there is no data available to be read at the time this method is invoked, the session thread will block until at least one byte of data becomes available, the timeout period elapses or an error occurs. This method will return if any amount of data is sent by the client, and will not block until the entire buffer has been filled.

The application should never make an assumption about the amount of data that will be available to read. TCP considers all data to be an arbitrary stream of bytes and does not impose any structure on the data itself. For example, if the client is sending data to the server in fixed 512 byte blocks of data, it is possible that a single call to the **Read** method will return only a partial block of data, or it may return multiple blocks combined together. It is the responsibility of the application to buffer and process this data appropriately.

For applications that are built using the Unicode character set, it is important to note that the buffer is an array of bytes, not characters. If the client is sending string data to the server, it must be read as a stream of bytes and converted using the **MultiByteToWideChar** function. If the client is sending lines of text terminated with a linefeed or carriage return and linefeed pair, the **ReadLine** method will return a line of text at a time and perform this conversion for you.

To determine if there is data available to be read without causing the session thread to block, call the **IsReadable** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[IsReadable](#), [IsWritable](#), [Peek](#), [ReadLine](#), [Write](#), [WriteLine](#)

CInternetServer::ReadLine Method

```
BOOL ReadLine(  
    SOCKET hSocket,  
    LPTSTR lpszBuffer,  
    LPINT lpnLength  
);  
  
BOOL ReadLine(  
    LPTSTR lpszBuffer,  
    LPINT lpnLength  
);  
  
BOOL ReadLine(  
    SOCKET hSocket,  
    CString& strBuffer,  
    INT nMaxLength  
);  
  
BOOL ReadLine(  
    CString& strBuffer,  
    INT nMaxLength  
);
```

Read up to a line of data from the socket and return it in a string buffer.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpszBuffer

Pointer to the string buffer that will contain the data when the method returns. The string will be terminated with a null character, and will not contain the end-of-line characters. An alternate form of the method accepts a **CString** argument which will contain the line of text returned by the server.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the method returns, its value will be updated with the actual length of the string.

nMaxLength

An integer value which specifies the maximum length of the buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ReadLine** method reads data from the socket and copies into a specified string buffer. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a

single line of text data in a string. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters. This method will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection.

There are some limitations when using **ReadLine**. The method should only be used to read text, never binary data. In particular, the method will discard nulls, linefeed and carriage return control characters. The Unicode version of this method will return a Unicode string, however it does not support reading raw Unicode data from the socket. Any data read from the socket is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

The **Read** and **ReadLine** method calls can be intermixed, however be aware that **Read** will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

Unlike the **Read** method, it is possible for data to be returned in the string buffer even if the return value is zero. Applications should check the length of the string to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the return value.

Example

```
CString strBuffer;
BOOL bResult;

do
{
    bResult = pServer->ReadLine(strBuffer);

    if (strBuffer.GetLength() > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = pServer->GetLastError();

if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The remote host has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[IsReadable](#), [Peek](#), [Read](#), [ReadStream](#), [Write](#), [WriteLine](#), [WriteStream](#)

CInternetServer::ReadStream Method

```
BOOL ReadStream(  
    SOCKET hSocket,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions,  
    LPBYTE lpMarker,  
    DWORD cbMarker  
);  
  
BOOL ReadStream(  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions,  
    LPBYTE lpMarker,  
    DWORD cbMarker  
);
```

Read a stream of data from the client and store it in the specified buffer.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpvBuffer

Pointer to the buffer that will contain or reference the data when the method returns. The actual argument depends on the value of the *dwOptions* parameter which specifies how the data stream will be stored.

lpdwLength

A pointer to an unsigned integer value which specifies the maximum length of the buffer and contains the number of bytes read when the method returns. This argument should always point to an initialized value. If the *lpvBuffer* argument specifies a memory buffer, then this argument cannot point to an initialized value of zero; if any other type of stream buffer is used and the initialized value is zero, that indicates that all available data from the socket should be returned until the end-of-stream marker is encountered or the remote host disconnects.

dwOptions

An unsigned integer value which specifies both the stream buffer type and any options to be used when reading the data stream.

Value	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the <i>lpvBuffer</i> parameter. If the argument specifies a pointer to a global memory handle initialized to NULL, then the method will return a handle which references the data; otherwise, the method will consider the parameter a pointer to a block of pre-allocated memory which will contain the stream data when the method returns. In most cases, it is recommended that an application explicitly specify the stream

	buffer type rather than using the default value.
INET_STREAM_MEMORY	The <i>lpvBuffer</i> argument specifies a pointer to a pre-allocated block of memory which will contain the data read from the socket when the method returns. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the maximum length of the buffer.
INET_STREAM_HGLOBAL	The <i>lpvBuffer</i> argument specifies a pointer to a global memory handle. When the method returns, the handle will reference a block of memory that contains the stream data. The application should take care to make sure that the handle passed to the method does not currently reference a valid block of memory; it is recommended that the handle be initialized to NULL prior to calling this method.
INET_STREAM_HANDLE	The <i>lpvBuffer</i> argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read from the socket will be written to this handle using the WriteFile function.
INET_STREAM_SOCKET	The <i>lpvBuffer</i> argument specifies a socket handle. The data read from the socket specified by the <i>hSocket</i> argument will be written to this socket. The socket handle passed to this method must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either create another instance of the class and attach the socket using the AttachHandle method or use the INET_STREAM_HANDLE stream buffer type instead.
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being returned as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the source data stream; if the application is using a pre-allocated memory buffer, this must be considered before calling this method.

A pointer to an array of bytes which marks the end of the data stream. When this byte sequence is encountered by the method, it will stop reading and return to the caller. The buffer will contain all of the data read from the socket up to and including the end-of-stream marker. If this argument is NULL, then the method will continue to read from the socket until the maximum buffer size is reached, the remote host closes its socket or an error is encountered.

cbMarker

An unsigned integer value which specifies the length of the end-of-stream marker in bytes. If the *lpMarker* parameter is NULL, then this value must be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ReadStream** method enables an application to read an arbitrarily large stream of data and store it in memory, write it to a file or even another socket. Unlike the **Read** method, which will return immediately when any amount of data has been read, **ReadStream** will only return when the buffer is full as specified by the *lpdwLength* parameter, the logical end-of-stream marker has been read, the socket closed by the remote host or when an error occurs. This method will force the session thread to block until the operation completes.

It is possible for data to be returned in the buffer even if the method returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Example

```
HGLOBAL hgblBuffer = NULL; // Return data in a global memory buffer
DWORD cbBuffer = 102400;   // Read up to 100K bytes
BOOL bResult;

bResult = pServer->ReadStream(&hgblBuffer, &cbBuffer,
                              INET_STREAM_HGLOBAL | INET_STREAM_CONVERT);

if (bResult && cbBuffer > 0)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgblBuffer);

    // Use data in the stream buffer

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

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: csoskv12.lib

See Also

CInternetServer::Reject Method

BOOL Reject();

Reject a pending client connection.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Reject** method rejects a pending client connection and the remote host will see this as the connection being aborted. If there are no pending client connections at the time, this method will immediately return with an error indicating that the operation would cause the server thread to block. This method should only be invoked from within an **OnAccept** event handler if the application wishes to reject the incoming connection before a client session is created.

To determine the IP address of a client that is attempting to connect to the server from within the **OnAccept** event, use the **GetClientAddress** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

See Also

[GetClientAddress](#), [OnAccept](#)

CInternetServer::Restart Method

BOOL Restart();

Restart the server, terminating all active client sessions.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Restart** method will restart the specified server, terminating all active client sessions. If the method is unable to restart the server for any reason, the server thread is terminated. The server retains all of the options specified for the previous instance.

If an application calls this method from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this method only be called by the same thread that created the server using the **Start** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[Start](#), [Stop](#)

CInternetServer::Resume Method

BOOL Resume();

Resume accepting client connections.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Resume** method instructs the server to resume accepting new client connections after the **Suspend** method has been called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[Restart](#), [Start](#), [Stop](#), [Suspend](#), [Throttle](#)

InternetServer::SetBacklog Method

```
BOOL SetBacklog(  
    UINT nBackLog  
);
```

Set the size of the backlog connection queue for the server.

Parameters

nBacklog

An integer value that specifies the size of the connection backlog queue.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero.

Remarks

The **SetBacklog** method specifies the size of the queue allocated for pending client connections. A value of zero specifies that the queue should be set to a reasonable default value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections. This method should only be called prior to invoking the **Start** method, it does not have any effect on an active server. To get the current size of the backlog queue, use the **GetBacklog** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSKV12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetBacklog](#), [Start](#), [Data Members](#)

CInternetServer::SetClientData Method

```
BOOL SetClientData(  
    SOCKET hClient,  
    LPVOID lpvData  
);  
  
BOOL SetClientData(  
    LPVOID lpvData  
);
```

Associate application defined data with the specified client session.

Parameters

hClient

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpvData

Pointer to the application defined data associated with the specified client session.

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the client session could not be modified. To get extended error information, call the **GetLastError** method.

Remarks

The **SetClientData** method is used to associate application defined data with a specific client session. This is typically used to associate a pointer to a data structure or a class instance with the client socket. The data can be retrieved using the **GetClientData** method.

You should never specify a pointer to a local variable or data structure that will go out of scope when the calling method exits. If you do this, the pointer will no longer be valid after the method exits and attempting to dereference that pointer at some later time can cause an exception to be thrown and terminate the program. You should always allocate a block of memory for the data using a method such as **HeapAlloc** or **LocalAlloc**. If you specify the address of a static or global data structure, you must use thread synchronization methods when dereferencing and modifying that structure.

Example

```
UINT *pnValue1 = new UINT;  
UINT *pnValue2 = NULL;  
  
*pnValue1 = 1234;  
  
if (!pServer->SetClientData(hClient, pnValue1))  
{  
    // Unable to associate the data with this session  
    return;  
}  
  
if (!pServer->GetClientData(hClient, (LPVOID *)&pnValue2))  
{  
    // Unable to retrieve the data associated with this session
```

```
        return;  
    }  
  
    // pnValue2 == pnValue1  
    printf("The value of the user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetClientData](#)

CInternetServer::SetClientMoniker Method

```
INT SetClientMoniker(  
    SOCKET hSocket,  
    LPCTSTR lpszMoniker  
);  
  
INT SetClientMoniker(  
    LPCTSTR lpszMoniker  
);
```

Associate a unique string alias with the specified client session.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpszMoniker

Pointer to a string which specifies the moniker for the specified client socket. If this parameter is NULL or specifies an empty string, a moniker will no longer be associated with the client session.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call the **GetLastError** method.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. The **GetClientMoniker** method will return the moniker that was previously assigned to the client, if any. To obtain the socket handle associated with a given moniker, use the **FindClient** method.

Monikers are not case-sensitive, and they must be unique so that no client socket for a particular server can have the same moniker. The maximum length for a moniker is 127 characters.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[FindClient](#), [GetClientHandle](#), [GetClientId](#), [GetClientMoniker](#)

InternetServer::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

InternetServer::SetOptions Method

```
BOOL SetOptions(  
    DWORD dwOptions  
);
```

Set one or more server options.

Parameters

dwOptions

An unsigned integer that specifies one or more option bit flags.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero.

Remarks

The **SetOptions** method sets the server options for the class instance. For a list of the available options, see [Server Option Constants](#). This method should only be called prior to invoking the **Start** method, it does not have any effect on an active server. The **GetOptions** method returns the options that are currently specified for the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetOptions](#), [Data Members](#)

CInternetServer::SetCertificate Method

```
BOOL SetCertificate(  
    DWORD dwProtocol,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPCTSTR lpszPassword  
);  
  
BOOL SetCertificate(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPCTSTR lpszPassword  
);  
  
BOOL SetCertificate(  
    LPCTSTR lpszCertName,  
    LPCTSTR lpszPassword  
);
```

Specify the server certificate that should be used with secure connections.

Parameters

dwProtocol

An optional bitmask of supported security protocols. If this parameter is not specified, then a default set of security protocols will be automatically selected. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default selection of security protocols will be used when establishing a connection. The TLS 1.2, TLS 1.1 and TLS 1.0 protocols will be negotiated with the client, in that order of preference. This option will always request the latest version of the preferred security protocols and is the recommended value.
SECURITY_PROTOCOL_SSL	Either SSL 2.0 or SSL 3.0 should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the client. If this is the only protocol specified, TLS will be excluded from the list of supported protocols. Note that SSL 2.0 has been deprecated and will never be used by default.
SECURITY_PROTOCOL_TLS	The TLS 1.0, 1.1 or 1.2 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the client. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some clients that reject any attempt to use the older SSL protocol and require that only TLS be used.

lpzCertStore

An optional string value that specifies the name of the certificate store that contains the server certificate. If the certificate is stored in the registry, this parameter must specify a valid local certificate store name. If the certificate is stored in a file, this parameter should specify the full path to the file that contains the certificate. If this parameter is omitted, the personal certificate store for the current process will be used.

lpzCertName

A string value that specifies the name of the certificate. This parameter is required and cannot be NULL. Either the common name or the name assigned to the certificate may be specified. In most cases, this will be the fully qualified domain name of the host that the server is running on.

lpzPassword

An optional string value that specifies a password associated with the server certificate. This parameter is usually only required when the *lpzCertStore* parameter specifies a certificate stored in a file. If the server certificate does not have a password associated with it, this parameter is omitted.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: csksv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[EnableSecurity](#), [SetOptions](#), [Start](#), [ValidateCertificate](#)

CInternetServer::SetPriority Method

```
INT SetPriority(  
    INT nPriority  
);
```

Set the priority assigned to the specified server.

Parameters

nPriority

An integer that specifies the server priority. It may be one of the following values:

Value	Description
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the method succeeds, the return value is the previous priority assigned to the server. If the method fails, the return value is INET_ERROR.

Remarks

The **SetPriority** method changes the current priority assigned to the specified server. Client connections that are accepted after this method is called will inherit the new priority as their default priority. Previously existing client connections will not be affected by this function.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: csoskv12.lib

See Also

[GetPriority](#)

InternetServer::SetStackSize Method

```
BOOL SetStackSize(  
    DWORD dwStackSize  
);
```

Change the initial size of the stack allocated for threads created by the server.

Parameters

dwStackSize

The amount of memory that will be committed to the stack for each thread created by the server. If this value is zero, a default stack size will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **SetStackSize** method changes the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change this value unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cssock12.lib

See Also

[GetStackSize](#), [Start](#)

InternetServer::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

Set the timeout interval used when waiting for a blocking operation to complete.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **SetTimeout** method sets the amount of time that the server will wait for data to become available to read, and the default timeout for blocking network operations for each client session. If this method is invoked before the server has started, it will change the default timeout period for the entire server. If this method is invoked within a server event handler, it will change the timeout period for the active client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[GetTimeout](#), [IsReadable](#), [IsWritable](#), [OnTimeout](#)

CInternetServer::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

Display a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Remarks

The **ShowError** method will display a modal message box with an error message that corresponds to the specified error code. All top-level windows belonging to the current thread will be disabled until the user responds to the message box. An application should only invoke this method from within the main UI thread, never from within a server event handler such as **OnConnect**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CInternetServer::Start Method

```
BOOL Start(  
    LPCTSTR lpszLocalHost,  
    UINT nLocalPort,  
    UINT nMaxClients,  
    DWORD dwOptions  
);  
  
BOOL Start(  
    LPCTSTR lpszLocalHost,  
    UINT nLocalPort,  
    DWORD dwOptions  
);  
  
BOOL Start(  
    UINT nLocalPort,  
    DWORD dwOptions  
);  
  
BOOL Start(  
    UINT nLocalPort  
);
```

The **Start** method begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread. All interaction with the server and its client sessions takes place inside the class event handlers.

Parameters

lpszLocalHost

A null-terminated string which specifies the local hostname or IP address address that the server should be bound to. If this parameter is omitted or specifies a NULL pointer an appropriate address will automatically be used. If a specific address is used, the server will only accept client connections on the network interface that is bound to that address.

nLocalPort

The port number the server should use to listen for client connections. The port number used by the application must be unique and multiple instances of a server cannot use the same port number. It is recommended that a port number greater than 5000 be used for private, application-specific implementations.

nMaxClients

The maximum number of client connections that can be established with the server. A value of zero specifies that there should not be any fixed limit on the number of active client connections. This value can be adjusted after the server has been created by calling the **Throttle** method.

dwOptions

An unsigned integer value that specifies one or more options to be used when creating an instance of the server. For a list of the available options, see [Server Option Constants](#). If this parameter is omitted, the default options for the server instance will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To

get extended error information, call **GetLastError**.

Remarks

In most cases, the *lpzLocalHost* parameter should be omitted or a NULL pointer. On a multihomed system, this will enable the server to accept connections on any appropriately configured network adapter. Specifying a hostname or IP address will limit client connections to that particular address. Note that the hostname or address must be one that is assigned to the local system, otherwise an error will occur.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the method will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The server instance is managed in another thread, and all interaction with the server and active client connections are performed inside the event handlers. To disconnect all active connections, close the listening socket and terminate the server thread, call the **Stop** method.

Example

```
// EchoServer implementation
class CEchoServer : public CInternetServer
{
    void OnRead(SOCKET hSocket)
    {
        // Read data sent by the client to the server
        BYTE ioBuffer[1024];
        INT nBytesRead = Read(hSocket, ioBuffer, sizeof(ioBuffer));

        // Send a copy of the data back to the client
        if (nBytesRead > 0)
            Write(hSocket, ioBuffer, nBytesRead);
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    CEchoServer myServer;

    // Start the server listening for connections on port 7000
    if (myServer.Start(7000))
    {
        TCHAR szCommand[128], *pszCommand;

        // Process commands entered by the user at the console
        while (TRUE)
        {
            if ((pszCommand = _fgetts(szCommand, 128, stdin)) == NULL)
                break;

            if (_tcsicmp(pszCommand, _T("quit")) == 0)
                break;
        }
    }
}
```

```
        // Stop the server and terminate all clients
        myServer.Stop();
    }

    return 0;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[EnumClients](#), [Restart](#), [Stop](#)

InternetServer::Stop Method

BOOL Stop();

Stop the server, terminating all active client sessions and releasing the resources that were allocated for the server.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, it will return a value of zero.

Remarks

The **Stop** method instructs the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the server handle to be opaque and never depend on it being a specific value.

If an application calls this method from within an event handler, the active client session (the client for which the event handler was invoked) may not get a disconnect notification. It is recommended that this method only be called by the same thread that created the server using the **Start** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: csoskv12.lib

See Also

[Restart](#), [Start](#)

CInternetServer::StoreStream Method

```
BOOL StoreStream(  
    LPCTSTR lpszFileName,  
    DWORD dwLength,  
    LPDWORD lpdwCopied  
    DWORD dwOffset,  
    DWORD dwOptions  
);
```

The **StoreStream** method reads the socket data stream and stores the contents in the specified file.

Parameters

lpszFileName

Pointer to a string which specifies the name of the file to create or overwrite.

dwLength

An unsigned integer which specifies the maximum number of bytes to read from the socket and write to the file. If this value is zero, then the method will continue to read data from the socket until the remote host disconnects or an error occurs.

lpdwCopied

A pointer to an unsigned integer value which will contain the number of bytes written to the file when the method returns.

dwOffset

An unsigned integer which specifies the byte offset into the file where the method will start storing data read from the socket. Note that all data after this offset will be truncated. A value of zero specifies that the file should be completely overwritten if it already exists.

dwOptions

An unsigned integer value which specifies one or more options. Programs can use a bitwise operator to combine any of the following values:

Value	Description
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data written to the file to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being written as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the

source data stream. If the <i>dwOffset</i> parameter has a value of zero, the Unicode byte order mark (BOM) will be written to the beginning of the file.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **StoreStream** method enables an application to read an arbitrarily large stream of data and store it in a file. This method is essentially a simplified version of the **ReadStream** method, designed specifically to be used with files rather than memory buffers or handles.

Example

```
// Store all data sent by the client in a file
DWORD dwCopied = 0;
BOOL bResult = pServer->StoreStream(lpszFileName, 0, &dwCopied, 0,
INET_STREAM_CONVERT);

// Close the client connection to the server
pServer->Disconnect();

if (bResult && dwCopied > 0)
{
    // Process the data has been written to the file
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[Read](#), [ReadLine](#), [ReadStream](#), [Write](#), [WriteLine](#), [WriteStream](#)

CInternetServer::Suspend Method

BOOL Suspend();

Suspend the server and reject new client connections.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Suspend** method instructs the server to suspend accepting new client connections. Any incoming client connections will be rejected with an error message indicating that the server is currently unavailable. To resume accepting client connections, call the **Resume** method. Suspending the server will have no effect on clients that have already established a connection with the server.

It is recommended that you only suspend a server if absolutely necessary, and only for brief periods of time. If you want to limit the number of active client connections or control the connection rate for clients, use the **Throttle** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

See Also

[Restart](#), [Resume](#), [Start](#), [Stop](#), [Throttle](#)

CInternetServer::Throttle Method

```
BOOL Throttle(  
    UINT nMaxClients,  
    UINT nMaxClientsPerAddress,  
    DWORD dwConnectionRate  
);
```

The **Throttle** method limits the number of active client connections, connections per address and connection rate.

Parameters

nMaxClients

A value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections. A value of -1 specifies that the maximum number of clients should not be changed.

nMaxClientsPerAddress

A value which specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is a limit of four client connections per address. A value of -1 specifies that the maximum number of clients should not be changed.

dwConnectionRate

A value which specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate. A value of -1 specifies that the connection rate should not be changed.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Throttle** method is used to limit the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **Start** method is called with the maximum number of clients set to 100, and then **Throttle** is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will accept incoming client connection requests. For example, setting this parameter to a value of 1000 would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of

time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being rejected.

It is recommended that you always specify conservative connection limits for your server application based on expected usage. Allowing an unlimited number of client connections can potentially expose the system to denial-of-service attacks and should never be done for servers that are accessible over the Internet.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSKV12.lib

See Also

[Restart](#), [Resume](#), [Start](#), [Suspend](#)

CIInternetServer::Unlock Method

```
BOOL Unlock();
```

Unlock the server, allowing other client threads to resume execution.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **Unlock** method releases the lock on the specified server and allows any blocked threads to resume execution. Only one server may be locked at any one time, and only the thread which established the lock can unlock the server.

Every time the **Lock** method is called, an internal lock counter is incremented, and the lock will not be released until the lock count drops to zero. This means that each call to the **Lock** method must be matched by an equal number of calls to the **Unlock** method. Failure to do so will result in the server becoming non-responsive as it remains in a locked state.

The program should always check the return value from this method, and should never assume that the lock has been released. If a potential deadlock situation is detected, this method will fail and return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[IsLocked](#), [Lock](#)

CInternetServer::Write Method

```
INT Write(  
    SOCKET hSocket,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

Send the contents of the specified buffer to to the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the client.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is `INET_ERROR`. To get extended error information, call the **GetLastError** method.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[IsReadable](#), [IsWritable](#), [Read](#), [ReadLine](#), [WriteLine](#)

CInternetServer::WriteLine Method

```
BOOL WriteLine(  
    SOCKET hSocket,  
    LPCTSTR lpszBuffer,  
    LPINT lpnLength  
);  
  
BOOL WriteLine(  
    LPCTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

Write a line of data to the client, terminated with a carriage-return and linefeed.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the remote host. All characters up to, but not including, the terminating null character will be written to the socket. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then only a carriage-return and linefeed are written to the socket.

lpnLength

A pointer to an integer value which will contain the number of characters written to the socket, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **WriteLine** method writes a line of text to the remote host and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **Write** method which writes arbitrary bytes of data to the socket, the **WriteLine** method is specifically designed to write a single line of text data from a null-terminated string. This method will force the session thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection.

There are some limitations when using **WriteLine**. This method should only be used to send text, never binary data. In particular, it will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this method will accept a Unicode string, however it does not support writing raw Unicode data to the socket. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written to the socket as a stream of bytes.

The **Write** and **WriteLine** methods can be safely intermixed.

Unlike the **Write** method, it is possible for data to have been written to the socket if the return value is

zero. For example, if a timeout occurs while the method is waiting to send more data to the remote host, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the *lpnLength* argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[IsWritable](#), [Read](#), [ReadLine](#), [ReadStream](#), [Write](#), [WriteStream](#)

CInternetServer::WriteStream Method

```
BOOL WriteStream(  
    SOCKET hSocket,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);  
  
BOOL WriteStream(  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

Write a stream of data to the client.

Parameters

hSocket

An optional parameter that specifies the handle to the client socket. If this parameter is omitted, the socket handle for the active client session will be used. If this method is called outside of a server event handler, the socket handle must be specified.

lpvBuffer

Pointer to the buffer that contains or references the data to be written to the socket. The actual argument depends on the value of the *dwOptions* parameter which specifies how the data stream will be accessed.

lpdwLength

A pointer to an unsigned integer value which specifies the size of the buffer and contains the number of bytes written when the method returns. This argument should should always point to an initialized value. If the *lpvBuffer* argument specifies a memory buffer or global memory handle, then this argument cannot point to an initialized value of zero.

dwOptions

An unsigned integer value which specifies the stream buffer type to be used when writing the data stream to the socket. One of the following stream types may be specified:

Value	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the <i>lpvBuffer</i> parameter. If the argument specifies a global memory handle, then the method will write the data referenced by that handle; otherwise, the method will consider the parameter a pointer to a block of memory which contains data to be written. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The <i>lpvBuffer</i> argument specifies a pointer to a block of memory which contains the data to be written to the socket. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned

	integer which has been initialized with the size of the buffer.
INET_STREAM_HGLOBAL	The <i>lpvBuffer</i> argument specifies a global memory handle that references the data to be written to the socket. The handle must have been created by a call to the GlobalAlloc or GlobalReAlloc function. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HANDLE	The <i>lpvBuffer</i> argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read using the ReadFile function with this handle will be written to the socket.
INET_STREAM_SOCKET	The <i>lpvBuffer</i> argument specifies a socket handle. The data read from the socket specified by this handle will be written to the socket specified by the <i>hSocket</i> parameter. The socket handle passed to this method must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either create another instance of the class and attach the socket using the AttachHandle method or use the INET_STREAM_HANDLE stream buffer type instead.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **WriteStream** method enables an application to write an arbitrarily large stream of data from memory or a file to the specified socket. Unlike the **Write** method, which may not write all of the data in a single method call, **WriteStream** will only return when all of the data has been written or an error occurs. This method will force the thread to block until the operation completes.

It is possible for some data to have been written even if the method returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was sent. For example, if a timeout occurs while the method is waiting to write more data, it will return zero; however, some data may have already been written to the socket prior to the error condition.

Example

```
CFile *pFile = new CFile();
DWORD dwLength = 0;

if (!pFile->Open(strFileName, CFile::modeRead | CFile::shareDenyWrite))
    return;

dwLength = pFile->GetLength();
```



```
if (dwLength > 0)
{
    BOOL bResult = pServer->WriteStream(
        pFile->m_hFile,
        &dwLength,
        INET_STREAM_HANDLE);
}

delete pFile;
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: csoskv12.lib

See Also

[Read](#), [ReadLine](#), [ReadStream](#), [StoreStream](#), [Write](#), [WriteLine](#)

CInternetServer Event Handlers

Method	Description
OnAccept	The client is attempting to establish a connection to the server
OnConnect	The client has established a connection to the server
OnDisconnect	The client has disconnected from the server
OnError	The event handler encountered an error when processing a client event
OnRead	The client has sent data to the server
OnTimeout	The client has not sent data within the specified timeout period
OnWrite	The client is ready to receive data from the server

CInternetServer::OnAccept Method

```
virtual void OnAccept(  
    SOCKET hSocket  
);
```

A virtual method that is invoked when a client attempts to connect to the server.

Parameters

hSocket

A handle to the server socket.

Return Value

None.

Remarks

The **OnAccept** event handler is invoked when a client attempts to connect to the server, but prior to the connection being accepted. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

This event only occurs before the server has checked the active client limits. This event is typically used to reject a connection based on some criteria established by the server, such as the IP address of the client attempting to make the connection. To obtain the IP address of the client that is attempting to connect to the server, use the **GetClientAddress** method using the server handle.

If this event handler is not implemented, the server will permit the client connection to complete. To reject the connection attempt, call the **Reject** method using the handle to the server socket. Rejecting the client connection within the **OnAccept** event handler may cause unexpected behavior by the client application because the connection process will not complete normally. Instead of rejecting the client connection within the **OnAccept** handler, it is recommended that most server applications implement an **OnConnect** event handler, perform any required checks and then gracefully disconnect the client using the **Disconnect** method if needed.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[Disconnect](#), [OnConnect](#), [OnDisconnect](#)

CInternetServer::OnConnect Method

```
virtual void OnConnect(  
    SOCKET hSocket,  
    UINT nClientId,  
    LPCTSTR lpszAddress,  
    UINT nPort  
);
```

A virtual method that is invoked after the client has connected to the server.

Parameters

hSocket

A handle to the client socket.

nClientId

An unsigned integer which uniquely identifies the client session.

lpszAddress

A string that specifies the IP address of the client. This address may either be in IPv4 or IPv6 format, depending on how the server was configured and the address the client used to establish the connection.

nPort

An integer that specifies the port number that the client socket is bound to.

Return Value

None.

Remarks

The **OnConnect** event handler is invoked after the client has connected to the server. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

This event only occurs after the server has checked the active client limits and the TLS handshake has been performed, if security has been enabled. If the server has been suspended, or the limit on the maximum number of client sessions has been exceeded, the server will reject the connection prior to this event handler being invoked.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[Disconnect](#), [OnAccept](#), [OnDisconnect](#)

CInternetServer::OnDisconnect Method

```
virtual void OnDisconnect(  
    SOCKET hSocket  
);
```

A virtual method that is invoked when a client disconnects from the server.

Parameters

hSocket

A handle to the client socket.

Return Value

None.

Remarks

The **OnDisconnect** event handler is invoked when a client disconnects from the server, immediately before the client session is terminated. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[Disconnect](#), [OnConnect](#)

CInternetServer::OnError Method

```
virtual void OnConnect(  
    SOCKET hSocket,  
    UINT nEventId,  
    DWORD dwError  
);
```

A virtual method that is invoked when the server encounters an error while handling a client request.

Parameters

hSocket

An unsigned integer which uniquely identifies the client session.

nEventId

An unsigned integer which identifies the client event that was being processed when the error occurred. For a list of event identifiers, see [Server Event Constants](#).

dwError

An unsigned integer value that specifies the error code.

Return Value

None.

Remarks

The **OnError** event handler is invoked whenever an error occurs while an event is being processed by the server. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

It is important to note that this event is not raised for every error that occurs. The event only occurs when another event is being processed and an unhandled error occurs that must be reported back to the server application. The following are some common situations in which this event handler may be invoked:

- A network error occurs when the client connection is being accepted by the server. This could be the result of an aborted connection or some other lower-level failure reported by the networking subsystem on the server.
- The server is configured to use implicit TLS but cannot obtain the security credentials required to create the security context for the session. Usually this indicates that the server certificate cannot be found, or the certificate does not have a private key associated with it. It could also indicate a general problem with the cryptographic subsystem where the client and server could not successfully negotiate a cipher suite.
- Network errors that may occur when attempting to buffer data sent by the client. This usually indicates that the connection to the client has been aborted, either because the client is not acknowledging the data that has been exchanged with the server, or the client has terminated abnormally. This event will not occur if the client terminates the connection normally.

In most situations where this event handler is invoked, the error is not recoverable and the only action that can be taken is to terminate the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[OnConnect](#), [OnDisconnect](#), [OnTimeout](#)

CInternetServer::OnRead Method

```
virtual void OnRead(  
    SOCKET hSocket  
);
```

A virtual method that is invoked when a client sends data to the server.

Parameters

hSocket

A handle to the client socket.

Return Value

None.

Remarks

The **OnRead** event handler is invoked when a client sends data to the server. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method. All server applications must implement an **OnRead** event handler to process the data sent by the client.

This event occurs whenever there is data available to be read from the client. The server application reads the data using the **Read** or **ReadLine** method, and can then send data back to the client using the **Write** or **WriteLine** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[Read](#), [ReadLine](#), [Write](#), [WriteLine](#), [OnWrite](#)

CInternetServer::OnTimeout Method

```
virtual void OnTimeout(  
    SOCKET hSocket  
);
```

A virtual method that is invoked when the client has not sent any data to the server within the timeout period.

Parameters

hSocket

A handle to the client socket.

Return Value

None.

Remarks

The **OnTimeout** event handler is invoked when a client has not sent any data to the server within the timeout period specified when the server was started. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method.

This event handler is typically used to monitor the amount of time that a client is idle. The default timeout period for the server is 20 seconds, which would cause this event handler to be invoked whenever a client has not sent any data to the server in the last 20 seconds. The server may take no action, or it may disconnect the client after it has been idle for an extended period of time. To get the total amount of time that the client has been idle, call the **GetClientIdleTime** method. Note that while the server timeout period is specified in seconds, the **GetClientIdleTime** method returns the client idle time in milliseconds.

The default implementation for this event handler is to take no action. It is recommended that most server applications disconnect clients that are inactive. For typical client-server implementations that use transitory connections (where the client sends a single request to the server, the server responds and the connection is terminated) the amount of time that a client should be permitted to remain idle should be relatively low, usually 60 seconds or less. For persistent connections where there are multiple requests issued by the client over the lifetime of the session, a longer idle timeout period may be preferable.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetClientIdleTime](#)

CInternetServer::OnWrite Method

```
virtual void OnWrite(  
    SOCKET hSocket  
);
```

A virtual method that is invoked when the client is ready to receive data.

Parameters

hSocket

A handle to the client socket.

Return Value

None.

Remarks

The **OnWrite** event handler is invoked when a client is ready to receive data. To implement an event handler, the application should create a class derived from the **CInternetServer** class, and then override this method. All server applications must implement an **OnRead** event handler to process the data sent by the client.

This event occurs immediately after the **OnConnect** event and if security is enabled, after the TLS handshake has completed. It is used to notify the server application that the client is ready to receive data, and may be used to send an initial message to the client, typically identifying the server that it has connected to. In most cases, the **OnWrite** event handler will only be invoked once over the lifetime of the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[Write](#), [WriteLine](#), [OnRead](#)

Internet Server Data Structures

- [INETSTREAMINFO](#)
- [INTERNET_ADDRESS](#)

INETSTREAMINFO

This structure contains information about the data stream being currently read or written.

```
typedef struct _INETSTREAMINFO
{
    DWORD    dwStreamThread;
    DWORD    dwStreamSize;
    DWORD    dwStreamCopied;
    DWORD    dwStreamMode;
    DWORD    dwStreamError;
    DWORD    dwBytesPerSecond;
    DWORD    dwTimeElapsed;
    DWORD    dwTimeEstimated;
} INETSTREAMINFO, *LPINETSTREAMINFO;
```

Members

dwStreamThread

Specifies the numeric ID for the thread that created the socket.

dwStreamSize

The maximum number of bytes that will be read or written. This is the same value as the buffer length specified by the caller, and may be zero which indicates that no maximum size was specified. Note that if this value is zero, the application will be unable to calculate a completion percentage or estimate the amount of time for the operation to complete.

dwStreamCopied

The total number of bytes that have been copied to or from the stream buffer.

dwStreamMode

A numeric value which specifies the stream operation that is current being performed. It may be one of the following values:

Value	Description
INET_STREAM_READ	Data is being read from the socket and stored in the specified stream buffer.
INET_STREAM_WRITE	Data is being written from the specified stream buffer to the socket.

dwStreamError

The last error that occurred when reading or writing the data stream. If no error has occurred, this value will be zero.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the operation is completed. This is based on the average number of bytes transferred per second and requires that a maximum stream buffer size be specified by the caller.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

See Also

[ReadStream](#), [StoreStream](#), [WriteStream](#)

INTERNET_ADDRESS

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
    INT    ipFamily;
    BYTE   ipNumber[16];
} INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the <i>ipNumber</i> array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the *ipNumber* array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the *ipFamily* member first to make sure that it is actually an IPv4 address.

Requirements

- Minimum Desktop Platform:** Windows 7 Service Pack 1
- Minimum Server Platform:** Windows Server 2008 R2 Service Pack 1
- Header File:** cswsock12.h

SocketTools Library Error Codes

Value	Constant	Description
0x80042711	ST_ERROR_NOT_HANDLE_OWNER	Handle not owned by the current thread
0x80042712	ST_ERROR_FILE_NOT_FOUND	The specified file or directory does not exist
0x80042713	ST_ERROR_FILE_NOT_CREATED	The specified file could not be created
0x80042714	ST_ERROR_OPERATION_CANCELED	The blocking operation has been canceled
0x80042715	ST_ERROR_INVALID_FILE_TYPE	The specified file type is invalid or not a regular file
0x80042716	ST_ERROR_INVALID_DEVICE	The specified file type is invalid or not a regular file
0x80042717	ST_ERROR_TOO_MANY_PARAMETERS	The maximum number of function parameters has been exceeded
0x80042718	ST_ERROR_INVALID_FILE_NAME	The specified file name contains invalid characters or is too long
0x80042719	ST_ERROR_INVALID_FILE_HANDLE	Invalid file handle passed to function
0x8004271A	ST_ERROR_FILE_READ_FAILED	Unable to read data from the specified file
0x8004271B	ST_ERROR_FILE_WRITE_FAILED	Unable to write data to the specified file
0x8004271C	ST_ERROR_OUT_OF_MEMORY	Out of memory
0x8004271D	ST_ERROR_ACCESS_DENIED	Access denied
0x8004271E	ST_ERROR_INVALID_PARAMETER	Invalid argument passed to function
0x8004271F	ST_ERROR_CLIPBOARD_UNAVAILABLE	The system clipboard is currently unavailable
0x80042720	ST_ERROR_CLIPBOARD_EMPTY	The system clipboard is empty or does not contain any text data
0x80042721	ST_ERROR_FILE_EMPTY	The specified file does not contain any data
0x80042722	ST_ERROR_FILE_EXISTS	The specified file already exists
0x80042723	ST_ERROR_END_OF_FILE	End of file
0x80042724	ST_ERROR_DEVICE_NOT_FOUND	The specified device could not be found
0x80042725	ST_ERROR_DIRECTORY_NOT_FOUND	The specified directory could not be found
0x80042726	ST_ERROR_INVALID_BUFFER	Invalid memory address passed to function
0x80042727	ST_ERROR_BUFFER_TOO_SMALL	The specified buffer is too small

0x80042728	ST_ERROR_NO_HANDLES	No more handles available to this process
0x80042733	ST_ERROR_OPERATION_WOULD_BLOCK	The specified operation would block the current thread
0x80042734	ST_ERROR_OPERATION_IN_PROGRESS	A network operation is currently in progress
0x80042735	ST_ERROR_ALREADY_IN_PROGRESS	The specified operation is already in progress
0x80042736	ST_ERROR_INVALID_HANDLE	Invalid handle passed to function
0x80042737	ST_ERROR_INVALID_ADDRESS	Invalid network address specified
0x80042738	ST_ERROR_INVALID_SIZE	Datagram is too large to fit in specified buffer
0x80042739	ST_ERROR_INVALID_PROTOCOL	Invalid network protocol specified
0x8004273A	ST_ERROR_PROTOCOL_NOT_AVAILABLE	The specified network protocol is not available
0x8004273B	ST_ERROR_PROTOCOL_NOT_SUPPORTED	The specified protocol is not supported
0x8004273C	ST_ERROR_SOCKET_NOT_SUPPORTED	The specified socket type is not supported
0x8004273D	ST_ERROR_INVALID_OPTION	The specified option is invalid or not supported
0x8004273E	ST_ERROR_PROTOCOL_FAMILY	The specified protocol family is invalid or not supported
0x8004273F	ST_ERROR_PROTOCOL_ADDRESS	The specified address is invalid for this protocol family
0x80042740	ST_ERROR_ADDRESS_IN_USE	The specified address is in use by another process
0x80042741	ST_ERROR_ADDRESS_UNAVAILABLE	The specified address cannot be assigned
0x80042742	ST_ERROR_NETWORK_UNAVAILABLE	The networking subsystem is unavailable
0x80042743	ST_ERROR_NETWORK_UNREACHABLE	The specified network is unreachable
0x80042744	ST_ERROR_NETWORK_RESET	Network dropped connection on remote reset
0x80042745	ST_ERROR_CONNECTION_ABORTED	Network connection aborted by local host
0x80042746	ST_ERROR_CONNECTION_RESET	Network connection aborted by remote host
0x80042747	ST_ERROR_OUT_OF_BUFFERS	No buffer space available
0x80042748	ST_ERROR_ALREADY_CONNECTED	Connection already established with remote host
0x80042749	ST_ERROR_NOT_CONNECTED	No connection established with remote host

0x8004274A	ST_ERROR_CONNECTION_SHUTDOWN	Unable to send or receive data after connection shutdown
0x8004274C	ST_ERROR_OPERATION_TIMEOUT	The specified operation has timed out
0x8004274D	ST_ERROR_CONNECTION_REFUSED	The connection has been refused by the remote host
0x80042750	ST_ERROR_HOST_UNAVAILABLE	The specified host is unavailable
0x80042751	ST_ERROR_HOST_UNREACHABLE	The specified host is unreachable
0x80042753	ST_ERROR_TOO_MANY_PROCESSES	Too many processes have initialized the networking subsystem
0x80042755	ST_ERROR_TOO_MANY_THREADS	Too many threads have been created by the current process
0x80042756	ST_ERROR_TOO_MANY_SESSIONS	Too many client sessions have been created by the current process
0x80042762	ST_ERROR_INTERNAL_FAILURE	An unexpected internal error has occurred
0x8004276B	ST_ERROR_NETWORK_NOT_READY	The networking subsystem is not available on this system
0x8004276C	ST_ERROR_INVALID_VERSION	This version of the operating system is not supported
0x8004276D	ST_ERROR_NETWORK_NOT_INITIALIZED	The networking subsystem has not been initialized
0x80042775	ST_ERROR_REMOTE_SHUTDOWN	The remote host has initiated a graceful shutdown sequence
0x80042AF9	ST_ERROR_INVALID_HOSTNAME	The specified hostname is invalid or could not be resolved
0x80042AFA	ST_ERROR_HOSTNAME_NOT_FOUND	The specified hostname could not be found
0x80042AFB	ST_ERROR_HOSTNAME_REFUSED	Unable to resolve hostname, request refused
0x80042AFC	ST_ERROR_HOSTNAME_NOT_RESOLVED	Unable to resolve hostname, no address for specified host
0x80042EE1	ST_ERROR_INVALID_LICENSE	The specified runtime license for this product is invalid
0x80042EE2	ST_ERROR_PRODUCT_NOT_LICENSED	This product is not licensed to perform this operation
0x80042EE3	ST_ERROR_NOT_IMPLEMENTED	This function has not been implemented on this platform
0x80042EE4	ST_ERROR_UNKNOWN_LOCALHOST	Unable to determine local host name
0x80042EE5	ST_ERROR_INVALID_HOSTADDRESS	Invalid host address specified
0x80042EE6	ST_ERROR_INVALID_SERVICE_PORT	Invalid service port number specified

0x80042EE7	ST_ERROR_INVALID_SERVICE_NAME	Invalid or unknown service name specified
0x80042EE8	ST_ERROR_INVALID_EVENTID	Invalid event identifier specified
0x80042EE9	ST_ERROR_OPERATION_NOT_BLOCKING	No blocking operation in progress on this socket
0x80042F45	ST_ERROR_SECURITY_NOT_INITIALIZED	Unable to initialize security interface for this process
0x80042F46	ST_ERROR_SECURITY_CONTEXT	Unable to establish security context for this session
0x80042F47	ST_ERROR_SECURITY_CREDENTIALS	Unable to open certificate store or establish security credentials
0x80042F48	ST_ERROR_SECURITY_CERTIFICATE	Unable to validate the certificate for this session
0x80042F49	ST_ERROR_SECURITY_DECRYPTION	Unable to decrypt data stream
0x80042F4A	ST_ERROR_SECURITY_ENCRYPTION	Unable to encrypt data stream
0x80042FA9	ST_ERROR_OPERATION_NOT_SUPPORTED	The specified operation is not supported
0x8004FAA	ST_ERROR_INVALID_PROTOCOL_VERSION	Invalid application protocol version specified
0x80042FAB	ST_ERROR_NO_SERVER_RESPONSE	No data returned from server
0x80042FAC	ST_ERROR_INVALID_SERVER_RESPONSE	Invalid data returned from server
0x80042FAD	ST_ERROR_UNEXPECTED_SERVER_RESPONSE	Unexpected response code returned from server
0x80042FAE	ST_ERROR_SERVER_TRANSACTION_FAILED	Server transaction failed
0x80042FAF	ST_ERROR_SERVICE_UNAVAILABLE	The service is currently unavailable
0x80042FB0	ST_ERROR_SERVICE_NOT_READY	The service is not ready, try again later
0x80042FB1	ST_ERROR_SERVER_RESYNC_FAILED	Unable to resynchronize with server
0x80042FB2	ST_ERROR_INVALID_PROXY_TYPE	Invalid proxy server type specified
0x80042FB3	ST_ERROR_PROXY_REQUIRED	Resource must be accessed through specified proxy
0x80042FB4	ST_ERROR_INVALID_PROXY_LOGIN	Unable to login to proxy server using specified credentials
0x80042FB5	ST_ERROR_PROXY_RESYNC_FAILED	Unable to resynchronize with proxy server
0x80042FB6	ST_ERROR_INVALID_COMMAND	Invalid command specified
0x80042FB7	ST_ERROR_INVALID_COMMAND_PARAMETER	Invalid command parameter specified
0x80042FB8	ST_ERROR_INVALID_COMMAND_SEQUENCE	Invalid command sequence specified
0x80042FB9	ST_ERROR_COMMAND_NOT_IMPLEMENTED	Specified command not implemented on this server

0x80042FBA	ST_ERROR_COMMAND_NOT_AUTHORIZED	Specified command not authorized for the current user
0x80042FBB	ST_ERROR_COMMAND_ABORTED	Specified command was aborted by the remote host
0x80042FBC	ST_ERROR_OPTION_NOT_SUPPORTED	The specified option is not supported on this server
0x80042FBD	ST_ERROR_REQUEST_NOT_COMPLETED	The current request has not been completed
0x80042FBE	ST_ERROR_INVALID_USERNAME	The specified username is invalid
0x80042FBF	ST_ERROR_INVALID_PASSWORD	The specified password is invalid
0x80042FC0	ST_ERROR_INVALID_ACCOUNT	The specified account name is invalid
0x80042FC1	ST_ERROR_ACCOUNT_REQUIRED	Account name has not been specified
0x80042FC2	ST_ERROR_INVALID_AUTHENTICATION_TYPE	Invalid or unsupported authentication protocol specified
0x80042FC3	ST_ERROR_AUTHENTICATION_REQUIRED	User authentication is required
0x80042FC4	ST_ERROR_PROXY_AUTHENTICATION_REQUIRED	Proxy authentication required
0x80042FC5	ST_ERROR_ALREADY_AUTHENTICATED	User has already been authenticated
0x80042FC6	ST_ERROR_AUTHENTICATION_FAILED	Unable to authenticate the specified user
0x80042FDB	ST_ERROR_NETWORK_ADAPTER	Unable to determine network adapter configuration
0x80042FDC	ST_ERROR_INVALID_RECORD_TYPE	Invalid record type specified
0x80042FDD	ST_ERROR_INVALID_RECORD_NAME	Invalid record name specified
0x80042FDE	ST_ERROR_INVALID_RECORD_DATA	Invalid record data specified
0x80042FDF	ST_ERROR_CONNECTION_OPEN	Data connection already established
0x80042FE0	ST_ERROR_CONNECTION_CLOSED	Server closed data connection
0x80042FE1	ST_ERROR_CONNECTION_PASSIVE	Data connection is passive
0x80042FE2	ST_ERROR_CONNECTION_FAILED	Unable to open data connection to server
0x80042FE3	ST_ERROR_INVALID_SECURITY_LEVEL	Data connection cannot be opened with this security setting
0x80042FE4	ST_ERROR_CACHED_TLS_REQUIRED	Data connection requires cached TLS session
0x80042FE5	ST_ERROR_DATA_READ_ONLY	Data connection is read-only
0x80042FE6	ST_ERROR_DATA_WRITE_ONLY	Data connection is write-only
0x80042FE7	ST_ERROR_END_OF_DATA	End of data
0x80042FE8	ST_ERROR_REMOTE_FILE_UNAVAILABLE	Remote file is unavailable
0x80042FE9	ST_ERROR_INSUFFICIENT_STORAGE	Insufficient storage on server
0x80042FEA	ST_ERROR_STORAGE_ALLOCATION	File exceeded storage allocation on

		server
0x80042FEB	ST_ERROR_DIRECTORY_EXISTS	The specified directory already exists
0x80042FEC	ST_ERROR_DIRECTORY_EMPTY	No files returned by the server for the specified directory
0x80042FED	ST_ERROR_END_OF_DIRECTORY	End of directory listing
0x80042FEE	ST_ERROR_UNKNOWN_DIRECTORY_FORMAT	Unknown directory format
0x80042FEF	ST_ERROR_INVALID_RESOURCE	Invalid resource name specified
0x80042FF0	ST_ERROR_RESOURCE_REDIRECTED	The specified resource has been redirected
0x80042FF1	ST_ERROR_RESOURCE_RESTRICTED	Access to this resource has been restricted
0x80042FF2	ST_ERROR_RESOURCE_NOT_MODIFIED	The specified resource has not been modified
0x80042FF3	ST_ERROR_RESOURCE_NOT_FOUND	The specified resource cannot be found
0x80042FF4	ST_ERROR_RESOURCE_CONFLICT	Request could not be completed due to the current state of the resource
0x80042FF5	ST_ERROR_RESOURCE_REMOVED	The specified resource has been permanently removed from this server
0x80042FF6	ST_ERROR_CONTENT_LENGTH_REQUIRED	Request must include the content length
0x80042FF7	ST_ERROR_REQUEST_PRECONDITION	Request could not be completed due to server precondition
0x80042FF8	ST_ERROR_UNSUPPORTED_MEDIA_TYPE	Request specified an unsupported media type
0x80042FF9	ST_ERROR_INVALID_CONTENT_RANGE	Content range specified for this resource is invalid
0x80042FFA	ST_ERROR_INVALID_MESSAGE_PART	Message is not multipart or an invalid message part was specified
0x80042FFB	ST_ERROR_INVALID_MESSAGE_HEADER	The specified message header is invalid or has not been defined
0x80042FFC	ST_ERROR_INVALID_MESSAGE_BOUNDARY	The multipart message boundary has not been defined
0x80042FFD	ST_ERROR_NO_FILE_ATTACHMENT	The current message part does not contain a file attachment
0x80042FFE	ST_ERROR_UNKNOWN_FILE_TYPE	The specified file type could not be determined
0x80042FFF	ST_ERROR_DATA_NOT_ENCODED	The specified data block could not be encoded
0x80043000	ST_ERROR_DATA_NOT_DECODED	The specified data block could not be decoded
0x80043001	ST_ERROR_FILE_NOT_ENCODED	The specified file could not be encoded

0x80043002	ST_ERROR_FILE_NOT_DECODED	The specified file could not be decoded
0x80043003	ST_ERROR_NO_MESSAGE_TEXT	The message does not contain any text characters
0x80043004	ST_ERROR_INVALID_MESSAGE_TEXT	The message contains invalid text characters
0x80043005	ST_ERROR_INVALID_ENCODING_TYPE	Invalid encoding type specified
0x80043006	ST_ERROR_INVALID_MESSAGE_NUMBER	Invalid message number specified
0x80043007	ST_ERROR_NO_RETURN_ADDRESS	No valid return address specified
0x80043008	ST_ERROR_NO_VALID_RECIPIENTS	No valid recipients specified
0x80043009	ST_ERROR_INVALID_RECIPIENT	The specified recipient address is invalid
0x8004300A	ST_ERROR_RELAY_NOT_AUTHORIZED	The specified domain is invalid or server will not relay messages
0x8004300B	ST_ERROR_MAILBOX_UNAVAILABLE	Specified mailbox is currently unavailable
0x8004300C	ST_ERROR_MAILBOX_READONLY	The selected mailbox cannot be modified
0x8004300D	ST_ERROR_MAILBOX_NOT_SELECTED	No mailbox has been selected
0x8004300E	ST_ERROR_INVALID_MAILBOX	Specified mailbox is invalid
0x8004300F	ST_ERROR_INVALID_DOMAIN	The specified domain name is invalid or not recognized
0x80043010	ST_ERROR_INVALID_SENDER	The specified sender address is invalid or not recognized
0x80043011	ST_ERROR_MESSAGE_NOT_DELIVERED	Message not delivered to any of the specified recipients
0x80043012	ST_ERROR_END_OF_MESSAGE_DATA	No more message data available to be read
0x80043013	ST_ERROR_INVALID_MESSAGE_SIZE	The specified message size is invalid
0x80043014	ST_ERROR_MESSAGE_NOT_CREATED	The message could not be created in the specified mailbox
0x80043015	ST_ERROR_NO_MORE_MAILBOXES	No more mailboxes exist on this server
0x80043016	ST_ERROR_INVALID_EMULATION_TYPE	The specified terminal emulation type is invalid
0x80043017	ST_ERROR_INVALID_FONT_HANDLE	The specified font handle is invalid
0x80043018	ST_ERROR_INVALID_FONT_NAME	The specified font name is invalid or unavailable
0x80043019	ST_ERROR_INVALID_PACKET_SIZE	The specified packet size is invalid
0x8004301A	ST_ERROR_INVALID_PACKET_DATA	The specified packet data is invalid
0x8004301B	ST_ERROR_INVALID_PACKET_ID	The unique packet identifier is invalid
0x8004301C	ST_ERROR_PACKET_TTL_EXPIRED	The specified packet time-to-live period has expired

0x8004301D	ST_ERROR_INVALID_NEWSGROUP	Invalid newsgroup specified
0x8004301E	ST_ERROR_NO_NEWSGROUP_SELECTED	No newsgroup selected
0x8004301F	ST_ERROR_EMPTY_NEWSGROUP	No articles in specified newsgroup
0x80043020	ST_ERROR_INVALID_ARTICLE	Invalid article number specified
0x80043021	ST_ERROR_NO_ARTICLE_SELECTED	No article selected in the current newsgroup
0x80043022	ST_ERROR_FIRST_ARTICLE	First article in current newsgroup
0x80043023	ST_ERROR_LAST_ARTICLE	Last article in current newsgroup
0x80043024	ST_ERROR_ARTICLE_EXISTS	Unable to transfer article, article already exists
0x80043025	ST_ERROR_ARTICLE_REJECTED	Unable to transfer article, article rejected
0x80043026	ST_ERROR_ARTICLE_TRANSFER_FAILED	Article transfer failed
0x80043027	ST_ERROR_ARTICLE_POSTING_DENIED	Posting is not permitted on this server
0x80043028	ST_ERROR_ARTICLE_POSTING_FAILED	Unable to post article to the server
0x80043029	ST_ERROR_INVALID_DATE_FORMAT	The specified date format is not recognized
0x8004302A	ST_ERROR_FEATURE_NOT_SUPPORTED	The specified feature is not supported
0x8004302B	ST_ERROR_INVALID_FORM_HANDLE	The specified form handle is invalid or a form has not been created
0x8004302C	ST_ERROR_INVALID_FORM_ACTION	The specified form action is invalid or has not been specified
0x8004302D	ST_ERROR_INVALID_FORM_METHOD	The specified form method is invalid or not supported
0x8004302E	ST_ERROR_INVALID_FORM_TYPE	The specified form type is invalid or not supported
0x8004302F	ST_ERROR_INVALID_FORM_FIELD	The specified form field name is invalid or does not exist
0x80043030	ST_ERROR_EMPTY_FORM	The specified form does not contain any field values
0x80043031	ST_ERROR_MAXIMUM_CONNECTIONS	The maximum number of client connections exceeded
0x80043032	ST_ERROR_THREAD_CREATION_FAILED	Unable to create a new thread for the current process
0x80043033	ST_ERROR_INVALID_THREAD_HANDLE	The specified thread handle is no longer valid
0x80043034	ST_ERROR_THREAD_TERMINATED	The specified thread has been terminated
0x80043035	ST_ERROR_THREAD_DEADLOCK	The operation would result in the current thread becoming deadlocked
0x80043036	ST_ERROR_INVALID_CLIENT_MONIKER	The specified moniker is not associated

		with any client session
0x80043037	ST_ERROR_CLIENT_MONIKER_EXISTS	The specified moniker has been assigned to another client session
0x80043038	ST_ERROR_SERVER_INACTIVE	The specified server is not listening for client connections
0x80043039	ST_ERROR_SERVER_SUSPENDED	The specified server is suspended and not accepting client connections
0x8004303A	ST_ERROR_NO_MESSAGE_STORE	No message store has been specified
0x8004303B	ST_ERROR_MESSAGE_STORE_CHANGED	The message store has changed since it was last accessed
0x8004303C	ST_ERROR_MESSAGE_NOT_FOUND	No message was found that matches the specified criteria
0x8004303D	ST_ERROR_MESSAGE_DELETED	The specified message has been deleted
0x8004303E	ST_ERROR_FILE_CHECKSUM_MISMATCH	The specified file checksums do not match
0x8004303F	ST_ERROR_FILE_SIZE_MISMATCH	The specified file sizes do not match
0x80043040	ST_ERROR_INVALID_FEED_URL	The news feed URL is invalid or specifies an unsupported protocol
0x80043041	ST_ERROR_INVALID_FEED_FORMAT	The internal format of the news feed is invalid
0x80043042	ST_ERROR_INVALID_FEED_VERSION	This version of the news feed is not supported
0x80043043	ST_ERROR_CHANNEL_EMPTY	There are no valid items found in this news feed
0x80043044	ST_ERROR_INVALID_ITEM_NUMBER	The specified channel item identifier is invalid
0x80043045	ST_ERROR_ITEM_NOT_FOUND	The specified channel item could not be found
0x80043046	ST_ERROR_ITEM_EMPTY	The specified channel item does not contain any data
0x80043047	ST_ERROR_INVALID_ITEM_PROPERTY	The specified item property name is invalid
0x80043048	ST_ERROR_ITEM_PROPERTY_NOT_FOUND	The specified item property has not been defined
0x80043049	ST_ERROR_INVALID_CHANNEL_TITLE	The channel title is invalid or has not been defined
0x8004304A	ST_ERROR_INVALID_CHANNEL_LINK	The channel hyperlink is invalid or has not been defined
0x8004304B	ST_ERROR_INVALID_CHANNEL_DESCRIPTION	The channel description is invalid or has not been defined

0x8004304C	ST_ERROR_INVALID_ITEM_TEXT	The description for an item is invalid or has not been defined
0x8004304D	ST_ERROR_INVALID_ITEM_LINK	The hyperlink for an item is invalid or has not been defined
0x8004304E	ST_ERROR_INVALID_SERVICE_TYPE	The specified service type is invalid
0x8004304F	ST_ERROR_SERVICE_SUSPENDED	Access to the specified service has been suspended
0x80043050	ST_ERROR_SERVICE_RESTRICTED	Access to the specified service has been restricted
0x80043051	ST_ERROR_INVALID_PROVIDER_NAME	The specified provider name is invalid or unknown
0x80043052	ST_ERROR_INVALID_PHONE_NUMBER	The specified phone number is invalid or not supported in this region
0x80043053	ST_ERROR_GATEWAY_NOT_FOUND	A message gateway cannot be found for the specified provider
0x80043054	ST_ERROR_MESSAGE_TOO_LONG	The message exceeds the maximum number of characters permitted
0x80043055	ST_ERROR_INVALID_PROVIDER_DATA	The request returned invalid or incomplete service provider data
0x80043056	ST_ERROR_INVALID_GATEWAY_DATA	The request returned invalid or incomplete message gateway data
0x80043057	ST_ERROR_MULTIPLE_PROVIDERS	The request has returned multiple service providers
0x80043058	ST_ERROR_PROVIDER_NOT_FOUND	The specified service provider could not be found
0x80043059	ST_ERROR_INVALID_MESSAGE_SERVICE	The specified message is not supported with this service type
0x8004305A	ST_ERROR_INVALID_MESSAGE_FORMAT	The specified message format is invalid
0x8004305B	ST_ERROR_INVALID_CONFIGURATION	One or more configuration options are invalid for this service
0x8004305C	ST_ERROR_SERVER_ACTIVE	The requested action is not permitted while the server is active
0x8004305D	ST_ERROR_SERVER_PORT_BOUND	Unable to obtain exclusive use of the specified local port
0x8004305E	ST_ERROR_INVALID_CLIENT_SESSION	The specified client identifier is invalid for this session
0x8004305F	ST_ERROR_CLIENT_NOT_IDENTIFIED	The specified client has not provided user credentials
0x80043060	ST_ERROR_INVALID_CLIENT_STATE	The requested action cannot be performed at this time
0x80043061	ST_ERROR_INVALID_RESULT_CODE	The specified result code is not valid for

		this protocol
0x80043062	ST_ERROR_COMMAND_REQUIRED	The specified command is required and cannot be disabled
0x80043063	ST_ERROR_COMMAND_DISABLED	The specified command has been disabled
0x80043064	ST_ERROR_COMMAND_SEQUENCE	The command cannot be processed at this time
0x80043065	ST_ERROR_COMMAND_COMPLETED	The previous command has completed
0x80043066	ST_ERROR_INVALID_PROGRAM_NAME	The specified program name is invalid or unrecognized
0x80043067	ST_ERROR_INVALID_REQUEST_HEADER	The request header contains one or more invalid values
0x80043068	ST_ERROR_INVALID_VIRTUAL_HOST	The specified virtual host name is invalid
0x80043069	ST_ERROR_VIRTUAL_HOST_NOT_FOUND	The specified virtual host does not exist
0x8004306A	ST_ERROR_TOO_MANY_VIRTUAL_HOSTS	Too many virtual hosts created for this server
0x8004306B	ST_ERROR_INVALID_VIRTUAL_PATH	The specified virtual path name is invalid
0x8004306C	ST_ERROR_VIRTUAL_PATH_NOT_FOUND	The specified virtual path does not exist
0x8004306D	ST_ERROR_TOO_MANY_VIRTUAL_PATHS	Too many virtual paths created for this server
0x8004306E	ST_ERROR_INVALID_TASK	The asynchronous task identifier is invalid
0x8004306F	ST_ERROR_TASK_ACTIVE	The asynchronous task has not finished
0x80043070	ST_ERROR_TASK_QUEUED	The asynchronous task has been queued
0x80043071	ST_ERROR_TASK_SUSPENDED	The asynchronous task has been suspended
0x80043072	ST_ERROR_TASK_FINISHED	The asynchronous task has finished
0x80043073	ST_ERROR_INVALID_ACCOUNT_UUID	The account unique identifier is invalid
0x80043074	ST_ERROR_INVALID_ACCOUNT_ID	The application account identifier is invalid
0x80043075	ST_ERROR_INVALID_PRODUCT_ID	The product identifier is invalid
0x80043076	ST_ERROR_INVALID_SERIAL_NUMBER	The product serial number is invalid
0x80043077	ST_ERROR_INVALID_APPID	The application identifier is invalid
0x80043078	ST_ERROR_INVALID_APIKEY	The specified authorization key is invalid
0x80043079	ST_ERROR_ACCOUNT_EXISTS	The application account identifier already exists
0x8004307A	ST_ERROR_ACCOUNT_NOT_CREATED	The application account identifier was not created
0x8004307B	ST_ERROR_ACCOUNT_NOT_FOUND	The application account identifier was not

		found
0x8004307C	ST_ERROR_ACCOUNT_NOT_EXPIRED	Access to this account has not expired
0x8004307D	ST_ERROR_ACCOUNT_NOT_UPDATED	The application account could not be updated
0x8004307E	ST_ERROR_ACCOUNT_EXPIRED	Access to this account has expired
0x8004307F	ST_ERROR_ACCOUNT_REVOKED	Access to this account has been revoked
0x80043080	ST_ERROR_APIKEY_NOT_CREATED	The application key could not be created
0x80043081	ST_ERROR_APIKEY_NOT_FOUND	The application key could not be found
0x80043082	ST_ERROR_APIKEY_NOT_EXPIRED	The application key has not expired
0x80043083	ST_ERROR_APIKEY_NOT_UNIQUE	The application key identifier is not unique
0x80043084	ST_ERROR_APIKEY_NOT_UPDATED	The application key could not be updated
0x80043085	ST_ERROR_APIKEY_NOT_DELETED	The application key could not be deleted
0x80043086	ST_ERROR_APIKEY_EXISTS	The application key already exists
0x80043087	ST_ERROR_APIKEY_EXPIRED	The application key has expired and must be refreshed
0x80043088	ST_ERROR_APIKEY_REVOKED	The application key has been revoked
0x80043089	ST_ERROR_APIKEY_APPID	The application was not found or was not specified
0x8004308A	ST_ERROR_INVALID_TOKEN	The access token is invalid or was not specified
0x8004308B	ST_ERROR_TOKEN_NOT_CREATED	The access token could not be created
0x8004308C	ST_ERROR_TOKEN_NOT_FOUND	The access token could not be found
0x8004308D	ST_ERROR_TOKEN_NOT_EXPIRED	The access token has not expired
0x8004308E	ST_ERROR_TOKEN_NOT_UPDATED	The access token was not updated
0x8004308F	ST_ERROR_TOKEN_NOT_DELETED	The access token could not be deleted
0x80043090	ST_ERROR_TOKEN_EXPIRED	The access token has expired and must be refreshed
0x80043091	ST_ERROR_TOKEN_REVOKED	The access token has been revoked
0x80043092	ST_ERROR_NO_APIKEYS_FOUND	No application keys found for this account
0x80043093	ST_ERROR_NO_TOKENS_FOUND	No access tokens found for this application key
0x80043094	ST_ERROR_NO_TOKENS_REVOKED	No access tokens have been revoked
0x80043095	ST_ERROR_INVALID_STORAGE_OBJECT	Invalid storage object identifier
0x80043096	ST_ERROR_STORAGE_OBJECT_READONLY	The storage object is read-only
0x80043097	ST_ERROR_STORAGE_OBJECT_EXPIRED	Access to the storage object has expired

0x80043098	ST_ERROR_STORAGE_OBJECT_SIZE	The storage object size exceeds storage limits
0x80043099	ST_ERROR_STORAGE_OBJECT_DIGEST	The storage object digest is invalid or cannot be computed
0x8004309A	ST_ERROR_STORAGE_OBJECT_EXISTS	A storage object with this label already exists
0x8004309B	ST_ERROR_STORAGE_OBJECT_MODIFIED	A storage object with this label has been modified
0x8004309C	ST_ERROR_STORAGE_OBJECT_NOT_OWNER	The current user is not the storage object owner
0x8004309D	ST_ERROR_STORAGE_OBJECT_NOT_FOUND	The specified storage object does not exist
0x8004309E	ST_ERROR_STORAGE_OBJECT_NOT_CREATED	The storage object was not created
0x8004309F	ST_ERROR_STORAGE_OBJECT_NOT_MODIFIED	The storage object was not modified
0x800430A0	ST_ERROR_STORAGE_OBJECT_NOT_RENAMED	The storage object was not renamed
0x800430A1	ST_ERROR_STORAGE_FOLDER_EMPTY	The storage folder does not contain any objects
0x800430A2	ST_ERROR_STORAGE_ACCOUNT_QUOTA	The storage account has exceeded its quota
0x800430A3	ST_ERROR_STORAGE_ACCOUNT_LIMIT	The storage account has exceeded its object limit
0x800430A4	ST_ERROR_INVALID_STORAGE_TYPE	The specified storage type is invalid
0x800430A5	ST_ERROR_INVALID_STORAGE_PROVIDER	The specified storage provider is not available
0x800430A6	ST_ERROR_INVALID_STORAGE_REGION	The specified storage region is not available
0x800430A7	ST_ERROR_INVALID_STORAGE_CONTAINER	The storage container does not exist or cannot be accessed
0x800430A8	ST_ERROR_INVALID_STORAGE_LABEL	The storage object label is invalid or undefined
0x800430A9	ST_ERROR_INVALID_QUEUE_HANDLE	The specified queue handle is invalid or the queue has been deleted
0x800430AA	ST_ERROR_INVALID_QUEUE_FILE	The specified file identifier is not valid for this queue
0x800430AB	ST_ERROR_QUEUE_RUNNING	The operation cannot be performed while the queue is running
0x800430AC	ST_ERROR_QUEUE_STOPPED	The operation cannot be performed when the queue has stopped
0x800430AD	ST_ERROR_QUEUE_EMPTY	There are no files in the specified queue
0x800430AE	ST_ERROR_QUEUE_PAUSED	The operation cannot be performed

		while the queue is paused
0x800430AF	ST_ERROR_QUEUE_LOCKED	The operation cannot be performed while the queue is locked
0x800430B0	ST_ERROR_FILE_NOT_QUEUED	The specified file cannot be found in the queue
0x800430B1	ST_ERROR_END_OF_QUEUE	There are no more files in the specified queue
0x800430B2	ST_ERROR_TOO_MANY_FILES	The maximum number of files have been queued for transfer
0x800430B3	ST_ERROR_NO_QUEUED_TRANSFER	No queued file transfer is currently in progress
0x800430B4	ST_ERROR_INVALID_X509_CERTIFICATE	The specified X.509 format certificate is invalid
0x800430B5	ST_ERROR_INVALID_PKCS12_CERTIFICATE	The specified PKCS 12 format certificate is invalid
0x800430B6	ST_ERROR_INVALID_CIPHER_SUITE	The specified cipher suite is invalid or unavailable
0x800430B7	ST_ERROR_DEPRECATED_CIPHER_SUITE	The specified cipher suite is insecure and has been deprecated
0x800430B8	ST_ERROR_INVALID_CERTIFICATE_CHAIN	The certificate chain could not be validated
0x800430B9	ST_ERROR_INVALID_PRIVATE_KEY	The private key for the certificate is invalid
0x800430BA	ST_ERROR_INVALID_API_SESSION	The application session identifier is invalid
0x800430BB	ST_ERROR_EXPIRED_API_SESSION	The application session identifier has expired
0x800430BC	ST_ERROR_INVALID_API_TOKEN	The application token for this session is invalid
0x800430BD	ST_ERROR_EXPIRED_API_TOKEN	The application token for this session has expired
0x800430BE	ST_ERROR_INVALID_API_AUTHID	The authorization token for this session is invalid
0x800430BF	ST_ERROR_INVALID_API_ENDPOINT	The endpoint for the specified request is invalid
0x800430C0	ST_ERROR_INVALID_API_PAYLOAD	The data submitted with the specified request is invalid
0x800430C1	ST_ERROR_UNKNOWN_SESSION_OWNER	The current session owner is unknown or no longer valid
0x800430C2	ST_ERROR_REVOKED_SESSION_AUTH	The authorization token for this session has been revoked
0x800430C3	ST_ERROR_INVALID_URL_SCHEME	The scheme for the specified URL is

		invalid or not supported
0x800430C4	ST_ERROR_INVALID_URL_HOST	The host name for the specified URL is invalid
0x800430C5	ST_ERROR_INVALID_URL_PORT	The port number for the specified URL is invalid
0x800430C6	ST_ERROR_INVALID_URL_PATH	The resource path for the specified URL is invalid
0x800430C7	ST_ERROR_INVALID_CONTENT_TYPE	The content type is invalid or not supported
0x800430C8	ST_ERROR_UNKNOWN_CONTENT_TYPE	The content type cannot be determined
0x800430C9	ST_ERROR_INVALID_CHARSET	The specified character set is invalid or not supported
0x800430CA	ST_ERROR_INVALID_CODEPAGE	The specified code page is invalid or not supported
0x800430CB	ST_ERROR_INVALID_HEADER_TYPE	The specified header type is invalid or not supported
0x800430CC	ST_ERROR_INVALID_HEADER_NAME	The specified header name is invalid or not permitted
0x800430CD	ST_ERROR_INVALID_HEADER_VALUE	The specified header value is invalid or not permitted
0x800430CE	ST_ERROR_HEADER_NOT_FOUND	The specified header value is undefined
0x800430CF	ST_ERROR_INVALID_DIGEST_TYPE	The specified message digest type is invalid
0x800430D0	ST_ERROR_INVALID_DIGEST_VALUE	The specified message digest value is invalid
0x800430D1	ST_ERROR_FILE_DIGEST_MISMATCH	The message digest does not match the specified file
0x800430D2	ST_ERROR_FILE_CONTENT_MISMATCH	The contents of the specified files are not identical
0x800430D3	ST_ERROR_FILE_TIME_MISMATCH	The timestamps for the specified files are not identical
0x800430D4	ST_ERROR_INVALID_TIMESTAMP	The timestamp value is invalid or undefined
0x800430D5	ST_ERROR_INVALID_UNICODE	The string contains invalid or undefined Unicode code points
0x800430D6	ST_ERROR_BUFFER_TOO_LARGE	The specified buffer size exceeds the maximum length allowed
0x800430D7	ST_ERROR_DIRECTORY_NOT_CREATED	Unable to create the specified path or directory
0x800430D8	ST_ERROR_INVALID_LOCALE	The specified locale identifier is invalid

0x800430D9	ST_ERROR_INVALID_LANGUAGE	The specified language name or identifier is invalid
0x800430DA	ST_ERROR_GENERAL_POLICY_VIOLATION	The operation was blocked due to a general policy violation
0x800430DB	ST_ERROR_POLICY_FEATURE_DISABLED	Local policy prevents the use of this feature
0x800430DC	ST_ERROR_POLICY_NO_ZONEINFO	Local policy has disabled security zone information
0x800430DD	ST_ERROR_POLICY_DOMAIN_BLOCKED	Local security policy does not permit access to this domain
0x800430DE	ST_ERROR_POLICY_RESOURCE_BLOCKED	Local security policy does not permit access to this resource
0x800430DF	ST_ERROR_POLICY_ADDRESS_BLOCKED	Local security policy does not permit use of this address
0x800430E0	ST_ERROR_POLICY_CREDENTIALS_RESTRICTED	Local policy does not permit the use of stored credentials
0x800430E1	ST_ERROR_POLICY_FILETYPE_RESTRICTED	Local policy does not permit access to this file
0x800430E2	ST_ERROR_POLICY_TRUST_FAILURE	The operation was prevented by local trust policy restrictions
0x800430E3	ST_ERROR_POLICY_RATE_LIMITED	The request exceeds local security policy connection or request limits
0x800430E4	ST_ERROR_POLICY_ENCRYPTION_REQUIRED	The request requires a secure connection
0x800430E5	ST_ERROR_WEBVIEW2_NOT_INSTALLED	The WebView2 control is not installed on this system
0x800430E6	ST_ERROR_INVALID_WEBVIEW2_VERSION	The WebView2 control installed on this system does not meet minimum version requirements
0x800430E7	ST_ERROR_WEBVIEW2_NOT_ATTACHED	An instance of the WebView2 control has not been attached to a parent window
0x800430E8	ST_ERROR_WEBVIEW2_NOT_FOUND	The WebView2 control window could not be found or has been destroyed
0x800430E9	ST_ERROR_NAVIGATION_RESTRICTED	Navigation has been restricted for this session
0x800430EA	ST_ERROR_SECURITY_STATUS_UNKNOWN	The security status for this session cannot be determined
0x800430EB	ST_ERROR_UNSAFE_URL_SCHEME	Access to this resource is restricted due to an unsafe URL scheme
0x800430EC	ST_ERROR_USER_PROFILE_NOT_CREATED	Unable to create the specified user session profile
0x800430ED	ST_ERROR_USER_PROFILE_LOCKED	The user profile has been locked and

		cannot be shared with another process
0x800430EE	ST_ERROR_RATE_LIMIT_EXCEEDED	Too many requests have been submitted to the server
0x800430EF	ST_ERROR_PROTOCOL_VERSION_REQUIRED	The requested operation requires a different protocol version
0x800430F0	ST_ERROR_INVALID_PROXY_URL	The specified proxy URL is invalid or not a supported proxy type
0x800430F1	ST_ERROR_INVALID_CREDENTIAL_TYPE	The specified credential type is invalid or undefined
0x800430F2	ST_ERROR_INVALID_CREDENTIAL_NAME	The specified credential name is invalid or cannot be found
0x800430F3	ST_ERROR_INVALID_CREDENTIAL_SECRET	The specified credential exists but does not contain a valid secret
0x800430F4	ST_ERROR_INVALID_PROVIDER_ID	The specified service provider ID is invalid or unknown
0x800430F5	ST_ERROR_INVALID_PROVIDER_GUID	The specified service provider GUID is invalid or unknown
0x800430F6	ST_ERROR_PROVIDER_NOT_SUPPORTED	The service provider is not supported in this context
0x800430F7	ST_ERROR_NO_PROVIDER_BASE_URL	A base connection URL must be specified for this service provider
0x800430F8	ST_ERROR_NO_PROVIDER_MODELS	The service provider does not support enumerating models
0x800430F9	ST_ERROR_PROVIDER_FORMAT_CHANGED	The response format returned by the provider was not recognized or is incompatible
0x800430FA	ST_ERROR_MODEL_NAME_REQUIRED	Provider requires a model name, no default model is defined
0x800430FB	ST_ERROR_INVALID_MODEL_NAME	The specified model name is not valid for this provider
0x800430FC	ST_ERROR_INVALID_MODEL_OBJECT	An invalid model object was returned by the provider
0x800430FD	ST_ERROR_MODEL_UNAVAILABLE	The specified model is unavailable for this session
0x800430FE	ST_ERROR_MODEL_NOT_FOUND	The requested model was not found or is not available from the provider
0x800430FF	ST_ERROR_API_VERSION_REQUIRED	An API version has not been specified and it is required by the provider
0x80043100	ST_ERROR_INVALID_API_VERSION	The specified API version is invalid or undefined
0x80043101	ST_ERROR_INVALID_REQUEST_TYPE	The request is not supported by the

		service provider
0x80043102	ST_ERROR_INVALID_PROJECT	The specified project identifier is invalid or unknown
0x80043103	ST_ERROR_SYSTEM_PROMPT_TOO_LONG	The system prompt exceeds the maximum length allowed
0x80043104	ST_ERROR_INVALID_MESSAGE_ID	The specified message identifier is invalid
0x80043105	ST_ERROR_INVALID_MESSAGE_ROLE	The role assigned to this message is invalid
0x80043106	ST_ERROR_TOO_MANY_MESSAGES	The request exceeds the maximum number of messages allowed
0x80043107	ST_ERROR_NO_MESSAGES_FOUND	There are no messages which match the specified criteria
0x80043108	ST_ERROR_END_OF_MESSAGES	There are no more messages available
0x80043109	ST_ERROR_NO_MESSAGE_HISTORY	There are no messages available

SocketWrench Class Library

A general purpose TCP/IP networking library for developing client and server applications.

Reference

- [Class Methods](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

Class Name	CSocketWrench
File Name	CSWSKV12.DLL
Version	12.0.3024.1286
LibID	99EEE601-E935-4266-B046-771A823DB47D
Import Library	CSWSKV12.LIB
Dependencies	None
Standards	RFC 768, RFC 791, RFC 793

Overview

At the core of all of the SocketTools networking libraries is the Windows Sockets API. This provides a low level interface for sending and receiving data over the Internet or a local intranet using the Transmission Control Protocol (TCP) and/or User Datagram Protocol (UDP). The SocketWrench class library provides a simpler interface to the Windows Sockets API, without sacrificing features or functionality. Using SocketWrench, you can easily create client and server applications while avoiding many of the mundane tasks and common problems that developers face when building Internet applications.

This class library supports secure connections using the TLS 1.2 and TLS 1.3 protocols and can also be used to create secure, customized server applications. Both implicit and explicit TLS connections are supported, enabling the class to work with a wide variety of client and server applications without requiring third-party encryption libraries.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

SocketWrench Class Methods

Class	Description
CSocketWrench	Constructor which initializes the current instance of the class
~CSocketWrench	Destructor which releases resources allocated by the class
Method	Description
Abort	Abort the connection and immediately close the socket
Accept	Accept a connection request from a remote host
AttachHandle	Attach the specified client handle to this instance of the class
AttachThread	Attach the specified socket to another thread
Cancel	Cancel a blocking operation
CompareAddress	Compare two IP addresses to determine if they are identical
Connect	Connect to the specified server
ConnectUrl	Connect to the specified server using a URL
CreateSecurityCredentials	Create a new security credentials structure
DeleteSecurityCredentials	Delete a previously created security credentials structure
DetachHandle	Detach the handle for the current instance of this class
DisableEvents	Disable asynchronous event notification
DisableSecurity	Disable secure communication with the remote host
DisableTrace	Disable logging of network function calls
Disconnect	Disconnect from the current server
EnableEvents	Enable asynchronous event notification
EnableSecurity	Enable secure communication with the remote host
EnableTrace	Enable logging of network function calls to a file
EnumNetworkAddresses	Return the list of network addresses that are configured for the local host
Flush	Flush the send and receive buffers
FormatAddress	Convert an IP address in binary format into a printable string
FreezeEvents	Suspend or resume event handling by the application
GetAdapterAddress	Return the IP or MAC assigned to the specified network adapter
GetAddress	Convert an IP address string to a binary format
GetAddressFamily	Return the address family for the specified IP address
GetDefaultHostFile	Return the fully qualified path name of the host file on the local system
GetErrorString	Return a description for the specified error code
GetExternalAddress	Return the external IP address assigned to the local system

GetFirstAlias	Return the first alias for the specified host name
GetHandle	Return the client handle used by this instance of the class
GetHostAddress	Return the IP address assigned to the specified hostname
GetHostFile	Return the name of the host file
GetHostName	Return the hostname assigned to the specified IP address
GetLastError	Return the last error code
GetLocalAddress	Return the local IP address and port number for a socket
GetLocalName	Return the hostname assigned to the local system
GetMemoryUsage	Return the amount of memory allocated for the network connection
GetNextAlias	Return the next alias for the specified host name
GetOption	Return the current socket options
GetPeerAddress	Return the IP address of the peer that the socket is connected to
GetPeerPort	Returns the remote port number used by the client to establish the connection
GetPhysicalAddress	Return the media access control (MAC) address for the primary network adapter
GetSecurityInformation	Return information about the security characteristics of a connection
GetServiceName	Return the service name associated with a specified port number
GetServicePort	Return the port number associated with a service name
GetStatus	Report what sort of socket operation is in progress
GetStreamInfo	Return information about the current stream read or write operation
GetTimeout	Return the timeout interval for blocking operations, in seconds
GetUrlHostName	Return the host name and port number specified in a URL
GetVpnAdapter	Return the name of the active Virtual Private Network adapter
HostNameToUnicode	Converts the canonical form of a host name to its Unicode version
InetEventProc	Callback method that processes events generated on the socket
IsAddressNull	Determine if the specified IP address is a null address
IsAddressRoutable	Determine if the specified IP address is routable over the Internet
IsBlocking	Determine if the socket is performing a blocking operation
IsClosed	Determine if the remote host has closed its socket
IsConnected	Determine if the socket is connected to a remote host
IsInitialized	Determine if the class has been successfully initialized
IsListening	Determine if the socket is listening for a connection
IsProtocolAvailable	Determine if the specified protocol and address family are supported
IsReadable	Determine if data can read from the socket without blocking
IsUrgent	Determine if there is any out-of-band data available to be read
IsWritable	Determine if data can be written to to the socket without blocking

Listen	Listen for client connections on the specified socket
MatchHostName	Match a host name against of list of addresses including wildcards
NormalizeHostName	Return the canonical form of a host name
Peek	Read data from the socket without removing it from the socket buffer
Read	Read data from the socket
ReadLine	Read a line of data from the socket, storing it in a string buffer
ReadStream	Read a stream of data from the socket
RegisterEvent	Register an event callback function
Reject	Reject a pending client connection
SetHostFile	Specify the name of an alternate host table
SetLastError	Set the last error code
SetOption	Set one or more options for the current socket
SetTimeout	Set the interval used when waiting for a blocking operation to complete
ShowError	Display a message box with a description of the specified error
Shutdown	Disable reception or transmission of data
StoreStream	Read a stream of data from the socket and store it in a file
ValidateCertificate	Validate the specified security certificate is installed on the local system
ValidateHostName	Validate the specified host name and return the resolved IP address
Write	Write data to the socket
WriteLine	Write a line of data to the socket, terminated with a carriage-return and linefeed
WriteStream	Write a stream of data to the socket

CSocketWrench::~CSocketWrench

`~CSocketWrench();`

The **CSocketWrench** destructor releases resources allocated by the current instance of the **CSocketWrench** object. It also uninitializes the library if there are no other concurrent uses of the class.

Remarks

When a **CSocketWrench** object goes out of scope, the destructor is automatically called to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all handles that were created for the connection are destroyed.

The destructor is not called explicitly by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cs wskv12.lib

See Also

[CSocketWrench](#)

CSocketWrench::Abort Method

BOOL Abort();

Immediately close the socket without waiting for any remaining data to be written out.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Abort** method should only be used when the connection must be closed immediately before the application terminates. This method should only be used to abort client connections and should not be used with passive (listening) sockets. Server applications that need to abort an incoming client connection should use the **Reject** method.

In most cases, the application should call the **Disconnect** method to gracefully close the connection to the remote host. Aborting the connection will discard any buffered data and may cause errors or result in unpredictable behavior.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[Cancel](#), [Disconnect](#), [Reject](#)

CSocketWrench::Accept Method

```
BOOL Accept(  
    SOCKET hSocket,  
    UINT nTimeout,  
    DWORD dwOptions  
);  
  
BOOL Accept(  
    CSocketWrench& swServer,  
    UINT nTimeout,  
    DWORD dwOptions  
);
```

The **Accept** method is used to accept a pending client connection.

This method has been deprecated and is included for backwards compatibility. Use the **CInternetServer** class to create a server application.

Parameters

hSocket

Handle to the listening socket. This argument may also reference a **CSocketWrench** object which is listening for connections. In either case, the server socket must have been created by calling the **Listen** method.

nTimeout

The number of seconds that the server will wait for a client connection before failing the operation. This value is used only for blocking connections.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
INET_OPTION_KEEPAIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.
INET_OPTION_INLINE	This option controls how urgent (out-of-band) data is handled when reading data from the socket. If set, urgent data is placed in the data stream along with non-urgent data.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle,

	leaving it open after the parent process has stopped the server.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the TLS protocol.
INET_OPTION_SECURE	This option determines if a secure connection is established with the remote host.
INET_OPTION_SECURE_FALLBACK	This option specifies the server should permit the use of less secure cipher suites for compatibility with legacy clients. If this option is specified, the server will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When a connection is accepted by the server, the original listening socket continues to listen for more connections. If no event notification window is specified, then **Accept** will block until a client attempts to connect to the server or the timeout period expires.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

CSocketWrench::AttachHandle Method

```
VOID AttachHandle(  
    SOCKET hSocket  
);  
  
VOID AttachHandle(  
    SOCKET hSocket,  
    DWORD dwProcessId  
);
```

The **AttachHandle** method attaches the specified socket handle to the current instance of the class.

Parameters

hSocket

The socket handle that will be attached to the current instance of the class object.

dwProcessId

The process ID for the process that currently owns the socket handle. This value may be zero to specify the current process.

Return Value

None.

Remarks

This method is used to attach a socket handle created outside of the class using the SocketWrench API. Once the socket handle is attached to the class, the other class member functions may be used with that socket. If the socket was created by a third-party library or the Windows Sockets API, then the handle will be automatically inherited by the library.

If a socket handle has already been created for the class, that handle will be released when the new handle is attached to the class object. If you want to prevent the previous socket connection from being terminated, you must call the **DetachHandle** method. Failure to release the detached handle may result in a resource leak in your application.

If the *dwProcessId* parameter specifies another process, the socket will be duplicated into the current process, attached to the current thread and the original socket handle will be closed in the other process. This enables an application to effectively take control of a connection created by another process. The original socket handle must be inheritable by the by the current process and must be an actual Windows socket handle, not a pseudo-handle. This functionality is only supported on Windows NT 4.0 and later versions of the operating system with the Microsoft TCP/IP stack. Note that Layered Service Providers (LSPs) may interfere with the ability to inherit handles across processes.

If the socket was created by another process, it is initialized by the library in a blocking state, even if was originally using asynchronous socket events. If the application requires that the socket use events, it must explicitly call **EnableEvents**. A program should never try to attach to a secure connection created by another process because the attached socket will not have the security context required to encrypt and decrypt the data exchanged with the remote host.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

See Also

[AttachThread](#), [DetachHandle](#), [GetHandle](#)

CSocketWrench::AttachThread Method

```
DWORD AttachThread(  
    DWORD dwThreadId  
);
```

The **AttachThread** method attaches the specified client handle to another thread.

Parameters

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the client handle.

Return Value

If the method succeeds, the return value is the thread ID of the previous owner. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

When a client handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in a client application to create the client handle, and then pass that handle to another worker thread. The **AttachThread** method can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the method, the original owner of the handle can be restored before the worker thread terminates.

This method should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **AttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **Cancel** method and then release the handle after the blocking method exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the client handle used by the class until the destructor is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[AttachHandle](#), [Cancel](#), [Connect](#), [DetachHandle](#), [Disconnect](#), [GetHandle](#)

CSocketWrench::Cancel Method

```
BOOL Cancel(  
    SOCKET hSocket  
);  
  
BOOL Cancel();
```

The **Cancel** method cancels any outstanding blocking socket operation, causing the blocking method to fail. The application may then retry the operation or terminate the connection.

Parameters

hSocket

An optional parameter that specifies the handle to the socket. If this parameter is omitted, the socket handle for the current class instance will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When the **Cancel** method is called, the blocking method will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

This method is typically called from within an event handler to signal that the current blocking operation should stop. It may also be used to cancel a blocking operation that is occurring on another thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

See Also

[Abort](#), [Disconnect](#), [IsBlocking](#)

CSocketWrench::CompareAddress Method

```
BOOL CompareAddress(  
    LPINTERNET_ADDRESS LpAddress1,  
    LPINTERNET_ADDRESS LpAddress2  
);  
  
BOOL CompareAddress(  
    LPCTSTR LpszAddress1,  
    LPCTSTR LpszAddress2  
);
```

Compare two IP addresses to determine if they are identical.

Parameters

lpAddress1

A pointer to an INTERNET_ADDRESS structure that contains the first IP address to be compared. An alternate version of this method accepts a string that specifies the IP address to be compared.

lpAddress2

A pointer to an INTERNET_ADDRESS structure that contains the second IP address to be compared. An alternate version of this method accepts a string that specifies the IP address to be compared.

Return Value

If the method succeeds and the two addresses are identical, the return value is non-zero. If the method fails or the two addresses are not identical, the return value is zero. If either parameter is NULL, or the address family for the two addresses are not the same, the last error code will be updated. If the addresses are valid and in the same address family, but are not identical, the last error code will be set to NO_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetHostAddress](#), [GetLocalAddress](#), [GetPeerAddress](#), [INTERNET_ADDRESS](#)

CSocketWrench::Connect Method

```

BOOL Connect(
    LPCTSTR lpszRemoteHost,
    UINT nRemotePort
);

BOOL Connect(
    LPCTSTR lpszRemoteHost,
    UINT nRemotePort,
    UINT nProtocol,
    UINT nTimeout,
    DWORD dwOptions,
    LPCTSTR lpszLocalAddress,
    UINT nLocalPort
);

```

The **Connect** method is used to establish a connection with a server.

Parameters

lpszHostName
A null-terminated string which specifies the host name or IP address of the system you want to connect with. This parameter cannot be a URL and must only specify the name of the remote host. If this parameter is NULL or an empty string, the method will fail with an error indicating the host name is invalid.

nRemotePort
The port number used to establish the connection. Valid port numbers range in value from 1 through 65535 and a value outside of this range will cause the function to fail. Port numbers in the range of 49152 and 65535 are referred to as dynamic ports and are generally reserved for private use by client applications. You cannot specify a port number of zero when establishing an outbound connection.

nProtocol
The protocol to be used when establishing the connection. This may be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout
The number of seconds to wait for a response before failing the current operation.

dwOptions
An unsigned integer used to specify one or more socket options. This parameter is constructed by using the bitwise Or operator with any of the following values:

Value	Description

INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_DONTROUTE	This option specifies default routing should not be used. This option should not be specified unless absolutely necessary.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the TLS protocol.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the <i>lpCredentials</i> parameter. If the <i>lpCredentials</i> parameter is NULL, the connection will default to using TLS 1.2 or later and the strongest cipher suites available.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a

	connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies that this instance of the class may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the class instance is synchronized across multiple threads.

lpszLocalAddress

A null-terminated string that specifies the local IP address that the socket should be bound to. If this parameter is NULL, then an appropriate address will automatically be used. A specific address should only be used if it is required by the application.

nLocalPort

The local port number that the socket should be bound to. If this parameter is set to zero, then an appropriate port number will automatically be used. A specific port number should only be used if it is required by the application.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The *lpszHostName* parameter must specify a valid host name or IP address. Host names are resolved into an IP address by first checking the local hosts file and if the name is not found, a name server query will be performed to determine the IP address. If the Unicode version of this function is called and the host name includes non-ASCII characters, the host name will be automatically converted to an ASCII compatible format. Refer to the **NormalizeHostName** method for more information. To establish a connection using a URL rather than a host name, use the **ConnectUrl** method.

This method will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling the **Connect** method in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

If you use the INET_OPTION_SECURE option to enable a secure connection, the connection will always use implicit TLS. This means a secure session will be initiated immediately after the socket connection has been established with the server. A common example of a service which uses implicit TLS is the HTTPS protocol. Another type of secure connection is one that uses explicit TLS. This is when the client establishes a normal (non-secure) connection with the server and then explicitly switches to using a secure connection, typically by sending a command. If the server you are connecting to requires explicit TLS, you should not specify the INET_OPTION_SECURE option. Instead, connect without this option and then call the **EnableSecurity** method when you are ready to initiate the TLS handshake.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the class instance is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call methods using that instance of the class. The ownership of the class instance may be transferred from one thread to another using the **AttachThread** method.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any method in that instance of the class, regardless of which thread created it. It is important to note that this option

disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the class instance is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same instance.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[ConnectUrl](#), [Disconnect](#), [EnableSecurity](#), [Read](#), [ReadLine](#), [RegisterEvent](#), [Write](#), [WriteLine](#)

ConnectUrl Method

```
BOOL ConnectUrl(  
    LPCTSTR lpszUrl,  
    UINT nTimeout,  
    DWORD dwOptions  
);
```

The **ConnectUrl** method is used to establish a TCP connection with a server using the information provided in a URL.

Parameters

lpszUrl

A null-terminated string which specifies a URL used when establishing the connection. This parameter cannot be NULL or point to an empty string. If a non-standard URI scheme is used, the port number must be explicitly specified or the method will fail. See the remarks below for more information on the format supported by this method.

nTimeout

The number of seconds to wait for the connection to complete before failing the current operation. This parameter is optional and if omitted or the value is zero, a default timeout period will be used.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is optional and if omitted, no additional options will be specified. This parameter value is constructed by using the bitwise Or operator with any of the following values:

Value	Description
INET_OPTION_KEEPAIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This option is not necessary for most connections, particularly when the client will not be connected to the server for an extended period of time.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will

	always be permitted. This option only affects secure connections using the TLS protocol.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The connection will always default to using TLS 1.2 or later and the strongest cipher suites available on the client platform. This option may be automatically enabled if the URL scheme specifies a service which requires a secure connection. See the remarks below for more information.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

Return Value

If the method succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

The **ConnectUrl** method provides a simplified interface which can be used to establish a connection using a URL. This method can only be used to establish connections using TCP and does not currently support the use of URLs to connect with a service which uses UDP. The general format of the URL should look like this:

```
[scheme]:// [[username : password] @] hostname [:port] / [path;parameters ...]
```

This method recognizes most standard URI schemes which use this format. The host name and port number specified in the URL will be used to establish a connection and the remaining information will be discarded. If the URL does not explicitly specify a port number, the default port number associated with the scheme will be used as the default value. For example, consider the following:

```
https://www.example.com
```

In this example, there is no port number specified; instead, the default port for the **https://** scheme

would be used, which is port 443. The host name **www.example.com** would be resolved into an IP address and the connection established on port 443. This method will also recognize a simpler format which only includes the host name and port number without a URI scheme, such as:

www.example.com:443

When used in this way, the port number must always be provided. Without a URI scheme or an explicit port number, the method cannot determine what port number should be used when establishing the connection. The same also applies if a custom, non-standard URI scheme is provided which is not recognized.

If the URI scheme specifies a secure protocol which requires implicit TLS, this method will automatically enable the `INET_OPTION_SECURE` option. For example, providing a URL which uses the **https://** scheme will automatically enable a secure connection regardless if the *dwOptions* parameter includes that option. If a URI scheme is used in conjunction with a port number associated with a secure service, security will also be enabled for that connection. For example:

http://www.example.com:443

The standard **http://** scheme is used which does not indicate a secure connection. However, since port 443 is the standard port designated for a secure HTTP connection, a secure connection will be enabled by default, even if `INET_OPTION_SECURE` has not been specified by the caller. Alternatively, if a custom port number is specified in the URL or the scheme is not recognized as one which requires implicit TLS, security options will not be automatically enabled for the connection.

The host name portion of the URL can be specified as either a domain name or an IP address. Because an IPv6 address can contain colon characters, you must enclose the entire address in bracket `[]` characters. If this is not done, this method will return an error indicating the port number is invalid. For example, the URL **https://[2001:db8:0:0:1::128]/** uses an IPv6 host address and this would be considered valid. Without the brackets, this URL would not be accepted.

Important: The URL provided to this method will only be used to establish a connection with a server. This is a general purpose method which does not enable support for any particular application protocol and all implementation details are the responsibility of your application. If you require higher-level support for a specific Internet protocol, the SocketTools API provides comprehensive collection of higher-level classes which can be used to access those services.

If you use the `INET_OPTION_SECURE` option to enable a secure connection, the connection will always use implicit TLS. This means a secure session will be initiated immediately after the socket connection has been established with the server. A common example of a service which uses implicit TLS is the HTTPS protocol. Another type of secure connection is one that uses explicit TLS. This is when the client establishes a normal (non-secure) connection with the server and then explicitly switches to using a secure connection, typically by sending a command. If the server you are connecting to requires explicit TLS, you should not specify the `INET_OPTION_SECURE` option. Instead, connect without this option and then call the **InetEnableSecurity** method when you are ready to initiate the TLS handshake.

To prevent this method from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **InetConnectUrl** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call methods using that handle. The ownership of the handle may be transferred from one thread to another using the

InetAttachThread method.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any method using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a method in the library, it must ensure that no other thread will call another method at the same time using the same socket handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[Connect](#), [Disconnect](#), [EnableSecurity](#), [GetUrlHostName](#), [Read](#), [RegisterEvent](#), [Write](#)

CSocketWrench::CreateSecurityCredentials Method

```
BOOL CreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);  
  
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);  
  
BOOL CreateSecurityCredentials(  
    LPCTSTR lpszCertName  
);
```

The **CreateSecurityCredentials** method creates security credentials for the client session.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows

	Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpzUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpzPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpzCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store

is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpzCertName

A null-terminated string which specifies the name of the certificate to use. The method will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the method will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the method will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call the **GetLastError** method.

Remarks

A client application only needs to call this method if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* parameter should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was

created, it must be provided in the *lpzPassword* parameter or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* parameters are optional values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters can be NULL or omitted.

Example

```
pSocket->CreateSecurityCredentials(lpzCertName);

bConnected = pSocket->Connect(lpzHostName,
                              INET_PORT_HTTP,
                              INET_PROTOCOL_TCP,
                              INET_TIMEOUT,
                              INET_OPTION_SECURE);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[Connect](#), [DeleteSecurityCredentials](#), [GetSecurityInformation](#), [ValidateCertificate](#),
[SECURITYCREDENTIALS](#)

CSocketWrench::CSocketWrench Method

CSocketWrench();

The **CSocketWrench** constructor initializes the class library and validates the license key at runtime.

Remarks

If the constructor fails to validate the runtime license, subsequent methods in this class will fail. If the product is installed with an evaluation license, then the application will only function on the development system and cannot be redistributed.

The constructor calls the **InetInitialize** function to initialize the library, which dynamically loads other system libraries and allocates thread local storage. If you are using this class within another DLL, it is important that you do not create or destroy an instance of the class from within the **DllMain** function because it can result in deadlocks or access violation errors. You should not declare static or global instances of this class within another DLL if it is linked with the C runtime library (CRT) because it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[~CSocketWrench](#), [IsInitialized](#)

CSocketWrench::DeleteSecurityCredentials Method

```
VOID DeleteSecurityCredentials();
```

The **DeleteSecurityCredentials** method releases the security credentials for the current connection.

Parameters

None.

Return Value

None.

Remarks

This method can be used to release the memory allocated to the client or server credentials after a secure connection has been terminated. The security credentials are released when the class destructor is called, so it is normally not required that the application explicitly call this method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[CreateSecurityCredentials](#), [ValidateCertificate](#)

CSocketWrench::DetachHandle Method

`SOCKET DetachHandle();`

The **DetachHandle** method detaches the socket handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the client handle associated with the current instance of the class object. If there is no active connection, the value `INVALID_SOCKET` will be returned.

Remarks

This method is used to detach a socket handle created by the class for use with the SocketWrench API. Once the socket handle is detached from the class, no other class member functions may be called. Note that the handle must be explicitly closed at some later point by the process or a resource leak will occur.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[AttachHandle](#), [GetHandle](#)

CSocketWrench::DisableEvents Method

BOOL DisableEvents();

The **DisableEvents** method disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **DisableEvents** method is used to disable event message posting for the specified socket handle. Although this will immediately prevent any new events from being generated, it is possible that messages could be waiting in the message queue. Therefore, an application must be prepared to handle client event messages after this method has been called.

This method is automatically called if the socket has event notification enabled, and the **Disconnect** method is called. The same issues regarding outstanding event messages also applies in this situation, requiring that the application handle event messages that may reference a socket handle that is no longer valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[EnableEvents](#), [RegisterEvent](#)

CSocketWrench::DisableSecurity Method

```
INT DisableSecurity();
```

The **DisableSecurity** method disables a secure session with the remote host.

Parameters

None.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **DisableSecurity** method disables a secure session, with subsequent calls to **Read** and **Write** sending and receiving unencrypted data. It is important to note that because this method sends a shutdown message to terminate the secure session, this may cause connection to be closed by the remote host.

This method does not close the socket. Use the **Disconnect** method to close the socket and release the resources allocated for the current session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[CreateSecurityCredentials](#), [DeleteSecurityCredentials](#), [EnableSecurity](#)

CSocketWrench::DisableTrace Method

BOOL DisableTrace();

The **DisableTrace** method disables the logging of network function calls.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, a value of zero is returned.

Remarks

This method will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this method is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[EnableTrace](#)

CSocketWrench::Disconnect Method

BOOL Disconnect();

Terminate the connection, closing the socket and releasing the memory allocated for the session.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Once the connection has been terminated, the class instance socket handle is no longer valid and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new connection is established. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[Abort](#), [Accept](#), [Connect](#), [DisableEvents](#), [EnableEvents](#), [Listen](#)

CSocketWrench::EnableEvents Method

```
BOOL EnableEvents(  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **EnableEvents** method enables event notifications using Windows messages.

This method has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **RegisterEvent** method to register an event handler which is invoked when an event occurs.

Parameters

- hEventWnd**
Handle to the window which will receive the socket notification messages. This parameter must specify a valid window handle. If a NULL handle is specified, event notification will be disabled.
- uEventMsg**
The message that is received when a network event occurs. This value must be greater than 1024.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **EnableEvents** method is used to request that notification messages be posted to the specified window whenever a network event occurs. This allows an application to monitor the status of different socket operations.

The **wParam** argument will contain the client handle, the low word of the **lParam** argument will contain the event ID, and the high word will contain any error code. If no error has occurred, the high word will always have a value of zero. The following events may be generated:

Value	Description
INET_EVENT_CONNECT	The connection to the remote host has completed. The high word of the lParam parameter should be checked, since this notification message will be posted if an error has occurred.
INET_EVENT_DISCONNECT	The remote host has closed the connection to the client. The client should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to be read by the calling process. No additional messages will be posted until the client has read at least some of the data. This event is only generated if the client is in asynchronous mode.
INET_EVENT_WRITE	The application can now send data to the remote host. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a

	blocking operation. This event is only generated if the client is in asynchronous mode.
INET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The application may attempt to retry the operation, or may disconnect from the remote host and report an error to the user.
INET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

To disable event notification, call the **DisableEvents** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[DisableEvents](#), [RegisterEvent](#)

CSocketWrench::EnableSecurity Method

```
BOOL EnableSecurity();  
  
BOOL EnableSecurity(  
    LPCTSTR lpszCertName  
);  
  
BOOL EnableSecurity(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName  
);
```

The **EnableSecurity** method enables a secure session with the remote host.

Parameters

lpszCertStore

A null-terminated string which specifies the name of certificate store.

lpszCertName

A null-terminated string which specifies the common name for the certificate that will be used.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **EnableSecurity** method enables a secure communications session with the remote host, automatically negotiating the encryption algorithm and validating the certificate. If the socket was created using the **Connect** method to establish a client connection, then **EnableSecurity** will initiate the handshake with the remote host to establish a secure session. If the **Accept** method was used to accept a connection from a client, then the method will block and wait for the remote host to initiate the handshake.

This method is useful if the application needs to establish an initial, non-secure connection to the remote host and then negotiate a secure connection at a later point. If the method succeeds, all subsequent calls to **Read** and **Write** to receive and send data will be encrypted.

If no arguments are specified, then the security credentials established with a previous call to **CreateSecurityCredentials** will be used. If a certificate name is specified, then the current security credentials will be updated to use that certificate.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[CreateSecurityCredentials](#), [DeleteSecurityCredentials](#), [DisableSecurity](#)

CSocketWrench::EnableTrace Method

```
BOOL EnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **EnableTrace** method enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

Return Value

If the method succeeds, the return value is non-zero. A return value of zero indicates an error. Failure typically indicates that the log file cannot be created.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this method, and is not associated with a specific class instance. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "0x". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxx:xxxx:xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[DisableTrace](#)

CSocketWrench::EnumNetworkAddresses Method

```
INT EnumNetworkAddresses(  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddressList,  
    INT nMaxAddresses  
);  
  
INT EnumNetworkAddresses(  
    LPINTERNET_ADDRESS lpAddressList,  
    INT nMaxAddresses  
);
```

The **EnumNetworkAddresses** method returns the list of network addresses that are configured for the local host.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this method. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return both IPv4 or IPv6 addresses assigned to the local host, depending on how the system is configured and which network interfaces are enabled. This option is only recommended for applications that require support for IPv6 connections.
INET_ADDRESS_IPV4	Return only the IPv4 addresses assigned to the local host. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Return only the IPv6 addresses assigned to the local host. All bytes in the <i>ipNumber</i> array are significant. This option is only recommended for those applications that require support for IPv6 connections.

lpAddressList

A pointer to an array of [INTERNET_ADDRESS](#) structures that will contain the IP address of each local network interface. This parameter may be NULL, in which case the method will only return the number of available addresses.

nMaxAddresses

Maximum number of addresses to be returned. If the *lpAddressList* parameter is NULL, this value must be zero.

Return Value

If the method succeeds, the return value is the number of network addresses that are configured for the local host. If the method fails, the return value is INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

If the *nAddressFamily* parameter is specified as INET_ADDRESS_ANY, the application must be prepared to accept IPv6 addresses returned by this method. On Windows Vista and later versions of the operating system, IPv6 support is enabled and the local network adapter will have IPv6 addresses assigned to them by default. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* parameter should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

This method will ignore addresses that are bound to a disabled interface, as well as those addresses bound to a virtual loopback interface. For example, although the loopback address 127.0.0.1 is a valid network address, it will not be included in list of addresses returned by this method.

The first IPv4 or IPv6 address returned by this method is typically the address assigned to the primary network adapter on the local system. However, your application should not depend on addresses being returned in any particular order. If the system has virtualization software installed, this method may also include the IP addresses assigned to any virtualized network adapters installed by that software.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: csoskv12.lib

See Also

[GetAdapterAddress](#), [GetHostAddress](#), [GetLocalAddress](#)

CSocketWrench::Flush Method

BOOL Flush();

The **Flush** method flushes the internal send and receive buffers used by the socket.

Parameters

None.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Flush** method will flush any data waiting to be read or written to the remote host . It is important to note that this method is not similar to flushing data to a disk file; it does not ensure that a specific block of data has been written to the socket. For example, you should never call this function immediately after calling the **Write** method or prior to calling the **Disconnect** method.

An application never needs to use the **Flush** method under normal circumstances. This method is only to be used when the application needs to immediately return the socket to an inactive state with no pending data to be read or written. Calling this function may result in data loss and should only be used if you understand the implications of discarding any data which has been sent by the remote host.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CSocketWrench::FormatAddress Method

```
INT FormatAddress(  
    LPINTERNET_ADDRESS LpAddress,  
    LPTSTR LpszAddress,  
    INT cchAddress  
);  
  
INT FormatAddress(  
    LPINTERNET_ADDRESS LpAddress,  
    CString& strAddress  
);
```

The **FormatAddress** method converts a numeric IP address to a printable string. The format of the string depends on whether an IPv4 or IPv6 address is specified.

Parameters

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure which specifies the numeric IP address that should be converted to a string.

lpszAddress

The buffer that will contain the formatted IP address. This buffer should be at least 46 characters in length. This may also reference a **CString** object which will contain the formatted address when the method returns.

cchAddress

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is the length of the IP address string. If the method fails, the return value is `INET_ERROR`, meaning that the IP address could not be converted into a string. Typically this indicates that the pointer to the `INTERNET_ADDRESS` structure is invalid, or the data does not specify a valid IP address family.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetHostAddress](#), [GetLocalAddress](#), [GetPeerAddress](#), [INTERNET_ADDRESS](#)

CSocketWrench::FreezeEvents Method

```
INT FreezeEvents(  
    BOOL bFreeze  
);
```

The **FreezeEvents** method is used to suspend and resume event handling by the application.

Parameters

bFreeze

A non-zero value specifies that event handling should be suspended by the client. A zero value specifies that event handling should be resumed.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

This method should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all socket events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if the program has suspended event handling, and four read events occur, once event handling is resumed only one of those read events will be posted to the client. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

Unicode: Implemented as Unicode and ANSI versions

See Also

[DisableEvents](#), [EnableEvents](#), [RegisterEvent](#)

CSocketWrench::GetAdapterAddress Method

```
INT GetAdapterAddress(  
    INT nAdapterIndex,  
    INT nAddressType,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);  
  
INT GetAdapterAddress(  
    INT nAdapterIndex,  
    INT nAddressType,  
    CString& strAddress  
);
```

Return the IP or MAC assigned to the specified network adapter.

Parameters

nAdapterIndex

An integer value that identifies the network adapter.

nAddressType

An integer value which specifies the type of address that should be returned:

Value	Description
INET_ADAPTER_IPV4	The address string will contain the primary IPv4 unicast address assigned to the network adapter.
INET_ADAPTER_IPV6	The address string will contain the primary IPv6 unicast address assigned to the network adapter.
INET_ADAPTER_MAC	The address string will contain the media access control (MAC) address assigned to the network adapter.

lpszAddress

A string buffer that will contain the IP or MAC address assigned to the adapter. This parameter cannot be NULL and it is recommended that it be at least 64 characters in length to provide enough space for any address type. An alternate form of the method accepts a **CString** argument which will contain the hostname.

nMaxLength

The maximum number of characters that can be copied into the string buffer, including the terminating null character. If the buffer is too small to store the complete address, this method will fail.

Return Value

If the method succeeds, the return value is the number of characters copied to the string buffer, not including the terminating null character. A return value of zero indicates that the requested address type has not been assigned to the adapter. If the method fails, the return value is INET_ERROR and this typically indicates that either the adapter index is invalid or the string buffer is not large enough to store the complete address. To get extended error information, call **GetLastError**.

Remarks

The **GetAdapterAddress** method will return the IPv4, IPv6 or MAC address assigned to a specific network adapter. The primary network adapter has an index value of zero, and it increments for each adapter that is configured on the local system.

The media access control (MAC) address is a 48 bit or 64 bit value that is assigned to each network interface and is used for identification and access control. All network devices on the same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This method returns the MAC address string as sequence of hexadecimal values separated by a colon. An example of a 48 bit MAC address would be "01:23:45:67:89:AB". Note that some virtual network adapters may not have a MAC address assigned to them, in which case this method would return zero.

This method will ignore network adapters that have been disabled, as well as those that are bound to a virtual loopback interface. If the system has dial-up networking or virtualization software installed, this method may also return IP addresses assigned to a virtualized network adapters installed by that software.

Example

```
// Display the IPv4 address assigned to each network adapter
for (INT nIndex = 0;; nIndex++)
{
    CString strAddress;

    if (pSocket->GetAdapterAddress(nIndex, INET_ADAPTER_IPV4, strAddress) ==
INET_ERROR)
        break;

    _tprintf(_T("Adapter %d: %s\n"), nIndex, szAddress);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[EnumNetworkAddresses](#), [GetLocalAddress](#), [GetLocalName](#)

CSocketWrench::GetAddress Method

```
INT GetAddress(  
    LPCTSTR lpszAddress,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress  
);  
  
INT GetAddress(  
    LPCTSTR lpszAddress,  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **GetAddress** method converts an IP address string to binary format.

Parameters

lpszAddress

A null-terminated string which specifies an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address specified by the *lpszAddress* parameter. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the value of the <i>lpszAddress</i> parameter.
INET_ADDRESS_IPV4	Specifies that the address should be in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero. If the <i>lpszAddress</i> parameter does not specify a valid IPv4 address string, this method will fail.
INET_ADDRESS_IPV6	Specifies that the address should be in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. If the <i>lpszAddress</i> parameter does not specify a valid IPv6 address string, this method will fail.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

If the *nAddressFamily* parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible that an IPv6 address string has been specified. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member

should always be specified as `INET_ADDRESS_IPV4` to ensure that only IPv4 addresses are returned and any attempt to specify an IPv6 address string would result in an error.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpszAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

Unicode: Implemented as Unicode and ANSI versions

See Also

[FormatAddress](#), [IsAddressNull](#), [IsAddressRoutable](#), [IsProtocolAvailable](#), [INTERNET_ADDRESS](#)

CSocketWrench::GetDefaultHostFile Method

```
INT GetDefaultHostFile(  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

```
INT GetDefaultHostFile(  
    CString& lpszFileName  
);
```

The **GetDefaultHostFile** method returns the fully qualified path name of the host file on the local system. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostNames** method. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

The location of the default host file depends on the operating system. For Windows 95/98 and Windows Me the file is stored in C:\Windows\hosts and for Windows NT and later versions the file is stored in C:\Windows\system32\drivers\etc\hosts. Regardless of platform, there is no filename extension and this file may or may not exist on a given system.

Parameters

lpszFileName

Pointer to a string buffer that will contain the fully qualified file name to the default host file. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the method will return the length of the string, not including the terminating null byte.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the method succeeds, the return value is length of the string. A return value of zero indicates that the default host file could not be determined for the current platform. To get extended error information, call **GetLastError**.

Remarks

This method only returns the default location of the host file and does not determine if the file actually exists. It is not required that a host file be present on the system.

The default host file is processed before performing a nameserver lookup when resolving a hostname into an IP address, or an IP address into a hostname.

To specify an alternate local host file, use the **SetHostFile** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cssock12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetHostAddress](#), [GetHostFile](#), [GetHostName](#), [SetHostFile](#)

CSocketWrench::GetErrorString Method

```
INT GetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT nMaxLength  
);  
  
INT GetErrorString(  
    DWORD dwErrorCode,  
    CString& strDescription  
);
```

The **GetErrorString** method is used to return a description of a specific error code. Typically this is used in conjunction with the **GetLastError** method for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length. An alternate form of the method accepts a **CString** variable which will contain the error description.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the method succeeds, the return value is the length of the description string. If the method fails, the return value is zero, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the method is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetLastError](#), [SetLastError](#), [ShowError](#)

CSocketWrench::GetExternalAddress Method

```
INT GetExternalAddress(  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress,  
    INT nMaxLength  
);  
  
INT GetExternalAddress(  
    INT nAddressFamily,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);  
  
INT GetExternalAddress(  
    INT nAddressFamily,  
    CString& strAddress  
);
```

The **GetExternalAddress** method returns the external IP address for the local system.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return any external IP address available. IPv4 addresses will be preferred over IPv6 for backwards compatibility.
INET_ADDRESS_IPV4	Specifies that the external address should be in IPv4 format.
INET_ADDRESS_IPV6	Specifies that the external address should be in IPv6 format. If the local system does not have IPv6 enabled, the method will fail.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the external IP address of the local host in binary form.

lpszAddress

A null-terminated string buffer that will contain the external IP address of the local host.

nMaxLength

The maximum length of the string that will contain the IP address when the method returns.

Return Value

If the method succeeds, the return value is zero. If the function fails, it will return INET_ERROR. To get extended error information, call the **GetLastError** method.

Remarks

The **GetExternalAddress** method returns the IP address assigned to the router that connects the local host to the Internet. This is typically used by an application executing on a system in a local network that uses a router which performs Network Address Translation (NAT). In that network configuration, the **GetLocalAddress** method will only return the IP address for the local system on the

LAN side of the network unless a connection has already been established to a remote host. The **GetExternalAddress** method can be used to determine the IP address assigned to the router on the Internet side of the connection and can be particularly useful for servers running on a system behind a NAT router.

An application should always check the return value in case there is an error. Never assume that the return value is always a valid address. The method may be unable to determine the external IP address for the local system for a number of reasons, particularly if it is behind a firewall or uses a proxy server that restricts access to external sites on the Internet. If the method is able to obtain a valid external address for the local host, that address will be cached by the library. If you are using a VPN (Virtual Private Network), this method will return the external IP address for the VPN adapter and not your local network connection. The **GetVpnAdapter** method can be used to obtain information about the active VPN adapter.

Calling this method may cause the current thread to block until the external IP address can be resolved and should never be used in conjunction with asynchronous socket connections. If you need to call this method in an application which uses asynchronous sockets, it is recommended that you create a background worker thread and call this method from within that thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[FormatAddress](#), [GetHostAddress](#), [GetLocalAddress](#), [GetPeerAddress](#), [GetVpnAdapter](#), [INTERNET_ADDRESS](#)

CSocketWrench::GetFirstAlias Method

```
BOOL GetFirstAlias(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszHostAlias,  
    INT nMaxLength  
);  
  
BOOL GetFirstAlias(  
    LPCTSTR lpszHostName,  
    CString& strHostAlias  
);
```

The **GetFirstAlias** method returns the first alias for the specified host name.

Parameters

lpszHostName

A null-terminated string which specifies the host name that you wish to return aliases for. This should be complete domain name.

lpszHostAlias

A string buffer which will contain the first alias for the specified host name. This string should be at least 64 bytes in length. This argument may also reference a **CString** object which will contain the host alias when the method returns.

nMaxLength

Maximum number of characters that can be copied into the *lpszHostAlias* string buffer, including the terminating null byte.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
CSocketWrench sockClient;  
CString strHostAlias;  
BOOL bResult;  
  
m_ctlListBox.ResetContent();  
  
bResult = sockClient.GetFirstAlias(m_strHostName, strHostAlias);  
if (bResult == FALSE)  
{  
    sockClient.ShowError();  
    return;  
}  
  
while (bResult)  
{  
    m_ctlListBox.AddString(strHostAlias);  
    bResult = sockClient.GetNextAlias(strHostAlias);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetHostAddress](#), [GetHostName](#), [GetNextAlias](#)

CSocketWrench::GetHandle Method

`SOCKET GetHandle();`

The **GetHandle** method returns the socket handle associated with the current instance of the class.

Parameters

None.

Return Value

This method returns the socket handle associated with the current instance of the class object. If there is no active connection, the value `INVALID_SOCKET` will be returned.

Remarks

This method is used to obtain the client handle created by the class for use with the SocketTools API.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[AttachHandle](#), [DetachHandle](#), [IsInitialized](#)

CSocketWrench::GetHostAddress Method

```
INT GetHostAddress(  
    LPCTSTR lpszHostName,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS LpAddress  
);  
  
INT GetHostAddress(  
    LPCTSTR lpszHostName,  
    LPINTERNET_ADDRESS LpAddress  
);
```

The **InetGetHostAddress** method resolves the specified host name into an IP address in binary format.

Parameters

lpszHostName

The name of the host to resolve; this may be a fully-qualified domain name or an IP address. This method recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on how the host name can be resolved. By default, a preference will be given for returning an IPv4 address. However, if the host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the specified host.

Return Value

If the method succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

This method can also be used to convert an address in dot notation to a binary format. If the method must perform a DNS lookup to resolve the hostname, the calling thread will block. To ensure future

compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the *ipFamily* member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

The *nAddressFamily* parameter is used to specify a preference for the type of address returned, however it is possible that a host may not have an IPv4 or IPv6 address record, in which case this function will fail. Although IPv4 is still the most common address used at this time, an application should not assume that because a given host name does not have an IPv4 address, that the host name is invalid.

If the *nAddressFamily* parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible for a host name to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpzHostName* parameter specifies an host that only has an IPv6 (AAAA) DNS record.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetHostName](#), [GetLocalAddress](#), [GetLocalName](#), [GetPeerAddress](#), [IsProtocolAvailable](#), [INTERNET_ADDRESS](#)

CSocketWrench::GetHostFile Method

```
INT GetHostFile(  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);  
  
INT GetHostFile(  
    CString& strFileName  
);
```

The **GetHostFile** method returns the name of the host file previously set using the **SetHostFile** method. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostNames** method.

Parameters

lpszFileName

Pointer to a string buffer that will contain the host file name. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the method will return the length of the string, not including the terminating null character.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the method succeeds, the return value is length of the string. A return value of zero indicates that no host file has been specified or the method was unable to determine the file name. To get extended error information, call **GetLastError**. If the last error is zero, this indicates that no host file name has been specified for the current thread. If the last error is non-zero, this indicates the reason that the method failed.

Remarks

This method only returns the name of the host file that is cached in memory for the current thread. The contents of the file on the disk may have changed after the file was loaded into memory. To reload the host file or clear the cache, call the **SetHostFile** method.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

The host file returned by this method may be different than the default host file for the local system. To determine the file name for the default host file, use the **GetDefaultHostFile** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetDefaultHostFile](#), [GetHostAddress](#), [GetHostName](#), [SetHostFile](#)

CSocketWrench::GetHostName Method

```
INT GetHostName(  
    LPINTERNET_ADDRESS LpAddress,  
    LPTSTR LpszHostName,  
    INT cchHostName  
);  
  
INT GetHostName(  
    LPINTERNET_ADDRESS LpAddress,  
    CString& strHostName  
);
```

The **GetHostName** method performs a reverse lookup, returning the host name associated with a given IP address.

Parameters

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure which specifies the IP address that should be resolved into a host name.

lpszHostName

The buffer that will contain the host name. It is recommended that this buffer be at least 256 characters in length to accommodate the longest possible fully qualified domain name.

cchHostName

The maximum number of characters that can be copied into the buffer.

Return Value

If the method succeeds, the return value is the length of the hostname. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

If the method must perform a reverse DNS lookup to resolve the IP address into a host name, the calling thread will block. This method requires that the host have a PTR record, otherwise it will fail. Because many hosts do not have a PTR record, calling this method frequently may have a negative impact on the overall performance of the application.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **IsProtocolAvailable** method. If an IPv6 stack is not installed, this method will fail if the *lpAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cssock12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetHostAddress](#), [GetLocalAddress](#), [GetLocalName](#), [GetPeerAddress](#), [IsProtocolAvailable](#), [INTERNET_ADDRESS](#)

CSocketWrench::GetLastError Method

```
DWORD GetLastError();  
  
DWORD GetLastError(  
    CString& strDescription  
);
```

Parameters

strDescription

A string which will contain a description of the last error code value when the method returns. If no error has been set, or the last error code has been cleared, this string will be empty.

Return Value

The return value is the last error code value for the current thread. Functions set this value by calling the **SetLastError** method. The Return Value section of each reference page notes the conditions under which the method sets the last error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **GetLastError** method immediately when a method's return value indicates that an error has occurred. That is because some methods call **SetLastError(0)** when they succeed, clearing the error code set by the most recently failed method.

Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

The description of the error code is the same string that is returned by the **GetErrorString** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetErrorString](#), [SetLastError](#), [ShowError](#)

CSocketWrench::GetLocalAddress Method

```
INT GetLocalAddress(
    INT nAddressFamily,
    LPINTERNET_ADDRESS lpAddress,
    UINT *lpnPort
);

INT GetLocalAddress(
    LPTSTR lpszAddress,
    INT nMaxLength
);

INT GetLocalAddress(
    CString& strAddress,
    UINT *lpnPort
);
```

The **GetLocalAddress** method returns the local IP address and port number for the current socket.

Parameters

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the type of connection that was established. If the <i>hSocket</i> parameter is INVALID_SOCKET, a preference will be given for returning an IPv4 address. However, if the local host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the local host. If there is no active connection, this function will attempt to determine the IP address of the local host assigned by the system. If the address is not required, this parameter may be NULL.

lpszAddress

A null-terminated string that will contain the IP address of the local host. If this version of the method is used, the IP address is converted to a string format using the **FormatAddress** method. The string should be able to store at least 46 characters to ensure that both IPv4 and IPv6

formatted addresses can be returned without the possibility of a buffer overrun. An alternate form of the method accepts a **CString** argument which will contain the local address.

lpnPort

A pointer to an unsigned integer that will contain the local port number. If there is an active connection, this parameter will be set to the local port that the socket is bound to. If there is no active connection, this parameter is ignored. If the local port number is not required, this parameter may be NULL.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

To ensure future compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the *ipFamily* member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

If the *nAddressFamily* parameter is specified as `INET_ADDRESS_UNKNOWN`, the application must be prepared to handle IPv6 addresses because it is possible for the local host to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as `INET_ADDRESS_IPV4` to ensure that only IPv4 addresses are returned.

If the system is connected to the Internet through a local network and/or uses a router that performs Network Address Translation (NAT), the **GetLocalAddress** method will return the local, non-routable IP address assigned to the local system. To determine the public IP address has been assigned to the system, you should use the **GetExternalAddress** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetExternalAddress](#), [GetHostAddress](#), [GetHostName](#), [GetLocalName](#), [GetPeerAddress](#), [INTERNET_ADDRESS](#)

CSocketWrench::GetLocalName Method

```
INT GetLocalName(  
    LPTSTR lpszHostName,  
    INT cchHostName  
);  
  
INT GetLocalName(  
    CString& strHostName  
);
```

The **GetLocalName** method returns the hostname assigned to the local system.

Parameters

lpszHostName

The buffer that will contain the hostname. This parameter cannot be NULL. An alternate form of the method accepts a **CString** argument which will contain the local hostname.

cchHostName

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is the length of the hostname. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetHostAddress](#), [GetHostName](#), [GetLocalAddress](#), [GetPeerAddress](#)

CSocketWrench::GetMemoryUsage Method

```
SIZE_T GetMemoryUsage();
```

Return the amount of memory allocated for the network connection.

Parameters

None.

Return Value

If the method succeeds, the return value is the number of bytes allocated. If the method fails, the return value is zero. Call the **GetLastError** method to determine the cause of the failure.

Remarks

This method returns the amount of memory allocated for the socket. It enumerates all heap allocations made by the process and returns the total number of bytes allocated for the current socket handle. This value only reflects the amount of memory explicitly allocated by this class library and does not reflect the total working set size of the process, or memory allocated by any other libraries. To determine the working set size for the process, refer to the Win32 **GetProcessWorkingSetSize** and **GetProcessMemoryInfo** functions.

This method forces the current thread into a locked state, and all network operations will block until the method returns. Because this method enumerates all heaps created for the process, it can be an expensive operation, particularly when a large number of connections have been established. Frequent use of this method can significantly degrade the performance of the application. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active network connections. If the value returned by the method remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

CSocketWrench::GetNextAlias Method

```
BOOL GetNextAlias(  
    LPTSTR lpszHostAlias,  
    INT nMaxLength  
);
```

```
BOOL GetNextAlias(  
    CString& strHostAlias  
);
```

The **GetNextAlias** method returns the next alias for the host name specified in the call to **GetFirstAlias**.

Parameters

lpszHostAlias

A string buffer which will contain the next alias for the specified host name. This string should be at least 64 bytes in length. This argument may also reference a **CString** object which will contain the host alias when the method returns.

nMaxLength

Maximum number of characters that can be copied into the *lpszHostAlias* string buffer, including the terminating null byte.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Example

```
CSocketWrench sockClient;  
CString strHostAlias;  
BOOL bResult;  
  
m_ctlListBox.ResetContent();  
  
bResult = sockClient.GetFirstAlias(m_strHostName, strHostAlias);  
if (bResult == FALSE)  
{  
    sockClient.ShowError();  
    return;  
}  
  
while (bResult)  
{  
    m_ctlListBox.AddString(strHostAlias);  
    bResult = sockClient.GetNextAlias(strHostAlias);  
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetFirstAlias](#), [GetHostAddress](#), [GetHostName](#)

CSocketWrench::GetOption Method

```
INT GetOption(  
    DWORD dwOption,  
    LPBOOL lpbEnabled  
);
```

The **GetOption** method is used to determine if a specific socket option has been enabled.

Parameters

dwOption

An unsigned integer used to specify one of the socket options. These options cannot be combined. The following values are recognized:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_KEEPAIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.

lpbEnabled

A pointer to a boolean flag. If the option is enabled, the flag is set to a non-zero value, otherwise it is set to a value of zero.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Requirements

- Minimum Desktop Platform:** Windows 7 Service Pack 1
- Minimum Server Platform:** Windows Server 2008 R2 Service Pack 1
- Header File:** cswsock12.h
- Import Library:** cswskv12.lib

See Also

[Connect](#), [SetOption](#)

CSocketWrench::GetPeerAddress Method

```
INT GetPeerAddress(  
    LPINTERNET_ADDRESS LpAddress,  
    UINT *LpnRemotePort  
);  
  
INT GetPeerAddress(  
    LPTSTR LpszAddress,  
    INT nMaxLength  
);  
  
INT GetPeerAddress(  
    CString& strAddress  
);
```

The **GetPeerAddress** method returns the peer IP address and remote port number for the specified socket.

Parameters

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the remote host that is connected to the socket.

lpnRemotePort

A pointer to an unsigned integer that will contain the port number of the remote host that is connected to the socket.

lpszAddress

A null-terminated string buffer that will contain the formatted IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length.

nMaxLength

The maximum number of characters that can be copied into the address buffer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

If this method is called by a server application in response to a `INET_EVENT_ACCEPT` event, it will return the IP address and port number for the client that is attempting to establish the connection. If the peer address is unavailable, the *ipFamily* member of the `INTERNET_ADDRESS` structure will be zero.

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

It is not recommended that you use the port number for anything other than informational and logging purposes. Server applications should not make any assumptions about the specific port

number or range of port numbers that a client is using when establishing a connection to the server. The ephemeral port number that a client is bound to can vary based on the client operating system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetHostAddress](#), [GetHostName](#), [GetLocalAddress](#), [GetLocalName](#), [GetPeerPort](#), [INTERNET_ADDRESS](#)

CSocketWrench::GetPhysicalAddress Method

```
BOOL GetPhysicalAddress(  
    LPTSTR lpszAddress,  
    UINT cchAddress  
);  
  
BOOL GetPhysicalAddress(  
    CString& strAddress  
);
```

Return the media access control (MAC) address for the primary network adapter.

Parameters

lpszAddress

A string buffer that will contain the address in a printable format when the function returns. This parameter cannot be NULL. An alternate form of the method accepts a **CString** argument which will contain the address.

cchAddress

The maximum number of characters that can be copied into the buffer, including the terminating null character.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetPhysicalAddress** method returns the media access control (MAC) address for the primary network adapter. This is a 48 bit or 64 bit address that is assigned to each network interface and is used for identification and access control. All network devices on the same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This method returns the MAC address as a printable string, with each byte of the address as a two-digit hexadecimal value separated by a colon. The string buffer passed to the method should be at least 20 characters long to accommodate the address and terminating null character. An example of a 48 bit address would be "01:23:45:67:89:AB". If the local system is multi-homed (having more than one network adapter) then this method will return the MAC address for the primary network adapter.

This method is provided for backwards compatibility with previous versions of the library and it is recommended that new applications use the **GetAdapterAddress** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[EnumNetworkAddresses](#), [GetAdapterAddress](#), [GetLocalName](#), [GetHostAddress](#)

CSocketWrench::GetSecurityInformation Method

```
BOOL GetSecurityInformation(  
    LPSECURITYINFO lpSecurityInfo  
);
```

The **GetSecurityInformation** method returns security protocol, encryption and certificate information about the current client connection.

Parameters

lpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this method.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

This method is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value SECURITY_PROTOCOL_NONE.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;  
securityInfo.dwSize = sizeof(SECURITYINFO);  
  
if (pSocket->GetSecurityInformation(&securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[Connect](#), [CreateSecurityCredentials](#), [EnableSecurity](#), [SECURITYINFO](#)

CSocketWrench::GetServiceName Method

```
BOOL GetServiceName(  
    UINT nServicePort,  
    LPTSTR lpszServiceName,  
    INT nMaxLength  
);  
  
BOOL GetServiceName(  
    UINT nServicePort,  
    CString& strServiceName  
);
```

The **GetServiceName** method returns the service name associated with a specified port number.

Parameters

nServicePort

Port number associated with some network service.

lpszServiceName

A null-terminated string buffer that will contain the service name when the method returns. This may also reference a **CString** object that will contain the service name.

nMaxLength

An integer value which specifies the maximum number of characters that can be copied into the string buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetServicePort](#)

CSocketWrench::GetServicePort Method

```
UINT GetServicePort(  
    LPCTSTR lpszServiceName  
);
```

The **GetServicePort** method returns the port number associated with a service name.

Parameters

lpszServiceName

A null-terminated string which specifies the name of the service to return the port number for.

Return Value

If the method succeeds, the return value is the port number associated with a service name. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetServiceName](#)

CSocketWrench::GetStatus Method

```
INT GetStatus();
```

The **GetStatus** method returns the current status of the socket.

Parameters

None.

Return Value

If the method succeeds, the return value is the client status code. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value is one of the following values:

Value	Description
INET_STATUS_UNUSED	No connection has been established.
INET_STATUS_IDLE	The socket is idle and not in a blocked state
INET_STATUS_LISTEN	The socket is listening for inbound connections from a client
INET_STATUS_CONNECT	The socket is establishing a connection with a server
INET_STATUS_ACCEPT	The socket is accepting a connection from a client
INET_STATUS_READ	Data is being read from the socket
INET_STATUS_WRITE	Data is being written to the socket
INET_STATUS_FLUSH	The socket is being flushed; all data in the receive buffers is being discarded
INET_STATUS_DISCONNECT	The socket is disconnecting from the remote host

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified socket.

Requirements

- Minimum Desktop Platform:** Windows 7 Service Pack 1
- Minimum Server Platform:** Windows Server 2008 R2 Service Pack 1
- Header File:** cswsock12.h
- Import Library:** cswskv12.lib

See Also

[IsConnected](#), [IsInitialized](#), [IsListening](#), [IsReadable](#), [IsWritable](#)

CSocketWrench::GetStreamInfo Method

```
BOOL GetStreamInfo(  
    LPINETSTREAMINFO LpStreamInfo  
);
```

The **GetStreamInfo** function fills a structure with information about the current stream I/O operation.

Parameters

LpSecurityInfo

A pointer to an [INETSTREAMINFO](#) structure which contains information about the status of the network stream.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetStreamInfo** method returns information about the current streaming socket operation, including the average number of bytes transferred per second and the estimated amount of time until the operation completes. If there is no operation currently in progress, this method will return the status of the last successful streaming read or write performed by the client.

In a multithreaded application, any thread in the current process may call this method to obtain status information for the specified socket.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[ReadStream](#), [StoreStream](#), [WriteStream](#), [INETSTREAMINFO](#)

CSocketWrench::GetTimeout Method

INT GetTimeout();

The **GetTimeout** method returns the number of seconds the client will wait for a response from the remote host. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

None.

Return Value

If the method succeeds, the return value is the timeout period in seconds. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections. A value of zero indicates that the default timeout period of 20 seconds should be used.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[Connect](#), [IsReadable](#), [IsWritable](#), [Read](#), [SetTimeout](#), [Write](#)

GetUrlHostName Method

```
INT GetUrlHostName(  
    LPCTSTR lpszUrl,  
    LPTSTR lpszHostName,  
    INT nMaxLength,  
    UINT *lpnHostPort,  
    UINT *lpnProtocol,  
    DWORD *lpdwOptions  
);
```

```
INT GetUrlHostName(  
    LPCTSTR lpszUrl,  
    CString& strHostName,  
    UINT *lpnHostPort,  
    UINT *lpnProtocol,  
    DWORD *lpdwOptions  
);
```

The **GetUrlHostName** method returns the host name and port number specified in a URL.

Parameters

lpszUrl

A null-terminated string which specifies a URL. This parameter cannot be NULL or point to an empty string. If a non-standard URI scheme is used, the port number must be explicitly specified or the method will fail. See the remarks below for more information on the format supported by this method.

lpszHostName

Pointer to the string buffer that will contain the canonical form of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer and must be large enough to store the complete host name. An alternate version of this method accepts a reference to a **CString** object if MFC or ATL is used with the project.

nMaxLength

The maximum number of characters that can be copied to the *lpszHostName* string buffer. This parameter cannot be zero, and must include the terminating null character.

lpnHostPort

Pointer to an optional unsigned integer value which will contain the port number specified in the URL. This parameter value will always be initialized by the method with a value of zero. If this parameter is omitted or NULL, it will be ignored and no port information will be returned.

lpnProtocol

Pointer to an optional unsigned integer value which will contain the protocol associated with the URI scheme. This parameter value will always be initialized by the method with a value of zero. If this parameter is omitted or NULL, it will be ignored and no protocol information will be returned.

lpdwOptions

Pointer to an optional unsigned integer value which will contain any socket options required to establish a connection based on the URI scheme or specified port. This parameter value will always be initialized by the method with a value of zero. If this parameter is omitted or NULL, it will be ignored.

Return Value

If the method succeeds, the return value is the number of characters copied into the *lpzHostName* buffer. If the method fails, the return value is `INET_ERROR`. To get extended error information, call the **GetLastError** method.

Remarks

The **GetUrlHostName** method will extract the host name and port number from a URL and return the canonical form of the host name. If the *lpnHostPort*, *lpnProtocol* and *lpdwOptions* parameters have been specified, they will contain the port number, protocol and additional connection options associated with the URL scheme.

The general format of the URL should look like this:

```
[scheme]:// [[username : password] @] hostname [:port] / [path;parameters ...]
```

This method recognizes most standard URI schemes which use this format. The host name and port number specified in the URL will be used to establish a connection and the remaining information will be discarded. If the URL does not explicitly specify a port number, the default port number associated with the scheme will be used as the default value. For example, consider the following:

```
https://www.example.com/
```

In this example, there is no port number specified; instead, the default port for the **https://** scheme would be used, which is port 443. This method will also recognize a simpler format which only includes the host name and port number without a URI scheme, such as:

```
www.example.com:443
```

If the *lpzUrl* parameter only specifies a host name without a URI scheme or port number, this method will ignore the *lpnHostPort*, *lpnProtocol* and *lpdwOptions* parameters and return the canonical form of the host name in the *lpzHostName* string argument.

The host name portion of the URL can be specified as either a domain name or an IP address. Because an IPv6 address can contain colon characters, you must enclose the entire address in bracket [] characters. If this is not done, the method will return an error indicating the port number is invalid. For example, the URL **https://[2001:db8:0:0:1::128]/** uses an IPv6 host address and this would be considered valid. Without the brackets, this URL would not be accepted.

If the URL uses an IP address instead of a host name, this method will return a copy of that IP address in the *lpzHostName* string provided by the caller. The method will not attempt to resolve the IP address into a host name, however you can use the **GetHostName** method to perform a reverse DNS lookup if required.

The only URI schemes currently supported by this method use TCP stream connections. In practical terms, this means the *lpnProtocol* parameter will always return with the value `INET_PROTOCOL_TCP` when the method is successful. If the method fails, this value will be `INET_PROTOCOL_NONE`.

Although this method performs checks to ensure that the *lpzUrl* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the host name. To check if the host name exists and has a valid IP address, use the **ValidateHostName** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csowskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[ConnectUrl](#), [GetHostAddress](#), [GetHostName](#), [HostNameToUnicode](#), [NormalizeHostName](#)

CSocketWrench::GetVpnAdapter Method

```
BOOL GetVpnAdapter(
    LPTSTR lpszAdapterName,
    INT nMaxLength,
    INT nAddressFamily,
    LPINTERNET_ADDRESS lpAddress
);

BOOL GetVpnAdapter(
    CString &strAdapterName,
    INT nAddressFamily,
    LPINTERNET_ADDRESS lpAddress
);
```

The **GetVpnAdapter** method returns the name of the active VPN adapter.

Parameters

lpszAdapterName

A null-terminated string buffer which will contain the name of the active VPN adapter if one is found. If a VPN was not detected, the buffer will contain an empty string when the method returns. This parameter may be NULL if the adapter name is not required. An alternate version of this method accepts a reference to a **CString** object which will contain the adapter name when the method returns.

nMaxLength

The maximum number of characters which can be copied into the string buffer, including the terminating null character. If this value is zero, the *lpszAdapterName* parameter is ignored.

nAddressFamily

An optional integer value which identifies the type of IP address to return. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return the IP address for the adapter in either IPv4 or IPv6 format, depending on how the host name can be resolved. By default, a preference will be given for returning an IPv4 address. However, if the adapter only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the adapter address should be returned in IPv4 format. The first four bytes of the <i>ipNumber</i> member of the INTERNET_ADDRESS structure are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the adapter address should be returned in IPv6 format. All bytes in the <i>ipNumber</i> member of the INTERNET_ADDRESS structure are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an `INTERNET_ADDRESS` structure that will contain the IP address assigned to the VPN adapter when the function returns. This parameter may be `NULL` or omitted if this information is not required. If this parameter is `NULL`, the value of the ***nAddressFamily*** parameter is ignored.

Return Value

If the method succeeds, the return value is non-zero if an active VPN adapter was found. If no VPN adapter was detected, this method will return zero.

Remarks

The **GetVpnAdapter** method provides a best-effort method to detect if the system is currently using a Virtual Private Network (VPN) connection. The detection logic is based on identifying active network adapters with names or descriptions that match known VPN software or common VPN adapter naming conventions, while explicitly excluding adapters associated with virtualization platforms (such as Hyper-V, VMware, and VirtualBox), loopback interfaces, and other non-VPN virtual adapters.

Because there is no industry standard for how VPN clients are identified at the network adapter level, this method uses a heuristic approach and cannot guarantee detection in every scenario. Some VPN clients may use custom adapter names or attempt to obscure their presence, and new VPN software may not be recognized until the internal detection list is updated. There is also a small chance that non-VPN software using similar naming conventions could result in a false positive.

The result of this method should be treated as a hint for diagnostic or troubleshooting purposes. It should not be used to restrict access to your application or deny service to users based solely on the possible presence of a VPN connection. If you specify an address family and the ***lpAddress*** parameter is not `NULL`, it will contain the first IP address bound to the adapter interface. It is possible for VPN adapters to be bound to multiple IP addresses. The **FormatAddress** method can be used to convert the `INTERNET_ADDRESS` value to a printable string.

The adapter name returned by this method is the description of the detected VPN adapter, encoded as a UTF-8 string in the ANSI version of this function. If you require more detailed network information, consider using the **EnumNetworkAddresses** method. You can use the **GetExternalAddress** method to obtain the public-facing IP address allocated for the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

Unicode: Implemented as Unicode and ANSI versions

See Also

[EnumNetworkAddresses](#), [FormatAddress](#), [GetAdapterAddress](#), [GetExternalAddress](#), [GetLocalAddress](#), [INTERNET_ADDRESS](#)

CSocketWrench::HostNameToUnicode Method

```
INT HostNameToUnicode(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszUnicodeName,  
    INT nMaxLength  
);  
  
INT HostNameToUnicode(  
    LPCTSTR lpszHostName,  
    CString& strUnicodeName  
);
```

The **HostNameToUnicode** method converts the canonical form of a host name to its Unicode version.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero-length string.

lpszUnicodeName

Pointer to the string buffer that will contain the original Unicode version of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer. An alternate version of this method accepts a reference to a **CString** object if MFC or ATL is used with the project.

nMaxLength

The maximum number of characters that can be copied to the *lpszUnicodeName* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **HostNameToUnicode** method will convert the encoded ASCII version of a host name to its Unicode version. Although any valid host name is accepted by this method, it is intended to convert a Punycode encoded host name to its original Unicode character encoding.

If the application is compiled using the Unicode character set, the value returned in *lpszUnicodeName* will be a Unicode string using UTF-16 encoding. If the ANSI character set is used, the value returned will be a Unicode string using UTF-8 encoding. To display a UTF-8 encoded host name, your application will need to convert it to UTF-16 using the **MultiByteToWideChar** function.

Although this method performs checks to ensure that the *lpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the [GetHostAddress](#) method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: csksv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetHostAddress](#), [NormalizeHostName](#)

CSocketWrench::IsAddressNull Method

```
BOOL IsAddressNull(  
    LPCTSTR lpszAddress  
);  
  
BOOL IsAddressNull(  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **IsAddressNull** method determines if the IP address is null.

Parameters

lpszAddress

A string that specifies the IP address.

lpAddress

A pointer to an INTERNET_ADDRESS structure that specifies the IP address.

Return Value

If the method succeeds and the IP address is null, or the parameter is a NULL pointer, the return value is non-zero. If the method fails or the address is not null, the return value is zero. If the address family is not supported, the last error code will be updated. If the address is valid but not null, the last error code will be set to NO_ERROR.

Remarks

A null IP address is one where all bits for the address (32 bits for IPv4 or 128 bits for IPv6) are zero. This is a special address that is typically used when creating a passive socket that should listen for connections on all available network interfaces.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetAddress](#), [IsAddressRoutable](#), [INTERNET_ADDRESS](#)

CSocketWrench::IsAddressRoutable Method

```
BOOL IsAddressRoutable(  
    LPCTSTR lpszAddress  
);  
  
BOOL IsAddressRoutable(  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **IsAddressRoutable** method determines if the IP address is routable over the Internet.

Parameters

lpszAddress

A string that specifies the IP address.

lpAddress

A pointer to an INTERNET_ADDRESS structure that specifies the IP address.

Return Value

If the method succeeds and the IP address is routable over the Internet, the return value is non-zero. If the method fails or the address is not routable, the return value is zero. If the parameter is NULL, or the address family is not supported, the last error code will be updated. If the address is valid but not routable, the last error code will be set to NO_ERROR.

Remarks

A routable IP address is one that can be reached by anyone over the public Internet. These are also commonly referred to as "public addresses" which are typically assigned to networks and individual hosts by an Internet service provider. There are also certain addresses that are not routable over the Internet, and used to address systems over a local network or private intranet. This function can be used to determine if a given IP address is public (routable) or private.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetAddress](#), [GetExternalAddress](#), [IsAddressNull](#), [INTERNET_ADDRESS](#)

CSocketWrench::IsBlocking Method

```
BOOL IsBlocking();
```

The **IsBlocking** method is used to determine if the socket is performing a blocking operation.

Parameters

None.

Return Value

If the socket is currently performing a blocking operation, the method returns a non-zero value. If the socket is not performing a blocking operation, or the socket handle is invalid, the method returns zero.

Remarks

This method is typically used to determine if a socket that is being used by another thread is currently blocked. A socket may block when waiting to receive data from a remote host or while data is actively being exchanged. Because there can only be one blocking socket operation per thread, this method can be used to determine if a method such as **Read** or **Write** would fail because another thread is currently sending or receiving data on that socket.

It is important to note that if this method returns a non-zero value, it does not guarantee that a subsequent read or write on the socket will succeed. The application should always check the return value from methods such as **Read** and **Write** to ensure they were successful.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[IsConnected](#), [IsReadable](#), [IsWritable](#), [Read](#), [ReadLine](#), [Write](#), [WriteLine](#)

CSocketWrench::IsClosed Method

```
BOOL IsClosed();
```

The **IsClosed** method is used to determine if the remote host has closed its socket.

Parameters

None.

Return Value

If the remote host has closed its socket, the method returns a non-zero value. If the remote host has not closed its connection, or the socket handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[IsConnected](#), [IsListening](#), [IsReadable](#), [IsWritable](#)

CSocketWrench::IsConnected Method

BOOL IsConnected();

The **IsConnected** method is used to determine if the socket is currently connected to a remote host.

Parameters

None.

Return Value

If the client is connected to a remote host, the method returns a non-zero value. If the client is not connected, or the client handle is invalid, the method returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetStatus](#), [IsBlocking](#), [IsInitialized](#), [IsReadable](#), [IsWritable](#)

CSocketWrench::IsInitialized Method

BOOL IsInitialized();

The **IsInitialized** method returns whether or not the class object has been successfully initialized.

Parameters

None.

Return Value

This method returns a non-zero value if the class object has been successfully initialized. A return value of zero indicates that the runtime license key could not be validated or the networking library could not be loaded by the current process.

Remarks

When an instance of the class is created, the class constructor will attempt to initialize the component with the runtime license key that was created when SocketTools was installed. If the constructor is unable to validate the license key or load the networking libraries, this initialization will fail.

If SocketTools was installed with an evaluation license, the application cannot be redistributed to another system. The class object will fail to initialize if the application is executed on another system, or if the evaluation period has expired. To redistribute your application, you must purchase a development license which will include the runtime key that is needed to redistribute your software to other systems. Refer to the Developer's Guide for more information.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[CSocketWrench](#), [IsBlocking](#), [IsConnected](#)

CSocketWrench::IsListening Method

BOOL IsListening();

The **IsListening** method determines if the socket is listening for connection requests.

Parameters

None.

Return Value

If the socket is being used to listen for connection requests, the method returns a non-zero value. If the socket is not listening or the socket handle is invalid, the method returns zero.

Remarks

The **IsListening** method determines if the socket is being used in a server application to actively listen for incoming connection requests from client applications. A listening socket can be created using the **Listen** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[IsReadable](#), [IsWritable](#), [IsConnected](#), [Listen](#)

CSocketWrench::IsProtocolAvailable Method

```
BOOL IsProtocolAvailable(  
    INT nAddressFamily,  
    INT nProtocol  
);
```

The **IsProtocolAvailable** method determines if the operating system supports creating a socket for the specified address family and protocol.

Parameters

nAddressFamily

An integer which identifies the address family that should be checked. It should be one of the following values:

Value	Description
INET_ADDRESS_IPV4	Specifies that the function should determine if it can create an Internet Protocol version 4 (IPv4) socket. This requires that the system have an IPv4 TCP/IP stack bound to at least one network adapter on the local system. All Windows systems include support for IPv4 by default.
INET_ADDRESS_IPV6	Specifies that the function should determine if it can create an Internet Protocol version 6 (IPv6) socket. This requires that the system have an IPv6 TCP/IP stack bound to at least one network adapter on the local system. Windows XP and Windows Server 2003 includes support for IPv6, however it is not installed by default. Windows Vista and later versions include support for IPv6 and enable it by default.

nProtocol

An integer which identifies the protocol that should be checked. It should be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This requires that the system be capable of creating a stream socket using the specified address family.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. This requires that the system be capable of creating a datagram socket using the specified address family.

Return Value

If the system is capable of creating a socket using the specified address family and protocol, this method will return a non-zero value. If the combination of address family and protocol is not supported, this method will return a value of zero.

Remarks

The **IsProtocolAvailable** method is used to determine if the operating system supports creating a particular type of socket. Typically it is used by an application to determine if the system has an IPv6 TCP/IP stack installed and configured. By default, all Windows systems will have an IPv4 stack installed if the system has a network adapter. However, not all systems may have an IPv6 stack installed, particularly older Windows XP and Windows Server 2003 systems. Note that if an IPv6 stack is not installed, the library will not recognize IPv6 addresses and cannot resolve host names that only have an IPv6 (AAAA) record, even if the address or host name is valid.

Example

```
if (!pSocket->IsProtocolAvailable(INET_ADDRESS_IPV6, INET_PROTOCOL_TCP))
{
    AfxMessageBox(_T("This system does not support IPv6"), MB_ICONEXCLAMATION);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetAddress](#), [GetHostAddress](#), [GetHostName](#)

CSocketWrench::IsReadable Method

```
BOOL IsReadable(  
    INT nTimeout,  
    LPDWORD lpdwAvail  
);
```

The **IsReadable** method is used to determine if data is available to be read from the remote host.

Parameters

nTimeout

Timeout for remote host response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read. This parameter may be NULL if this information is not required.

Return Value

If the current thread can read data from the socket without blocking, the method returns a non-zero value. If the current thread cannot read any data without blocking, the function returns zero.

Remarks

On some platforms, this value will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

If the connection is secure, the value returned in *lpdwAvail* will reflect the number of bytes available in the encrypted data stream. The actual amount of data available to the application after it has been decrypted will vary.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[IsWritable](#), [Peek](#), [Read](#), [ReadLine](#), [ReadStream](#)

CSocketWrench::IsUrgent Method

```
BOOL IsUrgent();
```

The **IsUrgent** method determines if there is any out-of-band (OOB) data available to be read.

Parameters

None.

Return Value

If there is out-of-band data, the return value is non-zero. If there is no out-of-band data, or an error occurs the return value is zero. To determine if an error has occurred, use the **GetLastError** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cswskv12.lib

See Also

[SetOption \(INET_OPTION_INLINE\)](#)

CSocketWrench::IsWritable Method

```
BOOL IsWritable(  
    INT nTimeout  
);
```

The **IsWritable** method is used to determine if data can be written to the remote host.

Parameters

nTimeout

Timeout for remote host response, in seconds. A value of zero specifies that the connection should be polled without blocking the current thread.

Return Value

If data can be written to the socket within the specified timeout period, the method returns a non-zero value. The method will return zero if the socket send buffer is full.

Remarks

The **IsWritable** method cannot be used to determine the amount of data that can be sent to the remote host without blocking the current thread. A non-zero return value only indicates that the send buffer is not full and can accept some data. In most cases, it is recommended that larger blocks of data be broken into smaller logical blocks rather than attempting to send it all of the data at once. For very large streams of data, it is recommended that you use the **WriteStream** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

See Also

[IsReadable](#), [Write](#), [WriteLine](#), [WriteStream](#)

CSocketWrench::Listen Method

```
SOCKET Listen(  
    LPCTSTR lpszLocalAddress,  
    UINT nLocalPort,  
    UINT nBacklog,  
    DWORD dwOptions  
);
```

The **Listen** method creates a listening socket and specifies the maximum number of connection requests that will be queued.

This method has been deprecated and is included for backwards compatibility. Use the **CInternetServer** class to create a server application.

Parameters

lpszLocalAddress

A null-terminated string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

nBacklog

The maximum length of queue of pending connections. On Windows workstations, the maximum backlog value is 5. On Windows servers, the maximum value is 200.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are recognized:

Value	Description
INET_OPTION_NONE	No option specified. If the address and port number are in use by another application or a closed socket which was listening on this port is still in the TIME_WAIT state, the function will fail.
INET_OPTION_REUSEADDRESS	This option enables a server application to listen for connections using the specified address and port number even if they were in use recently. This is typically used to enable an application to close the listening socket and immediately reopen it without getting an error that the address is in use.
INET_OPTION_EXCLUSIVE	This option specifies the local address and port number is for the exclusive use by the current process, preventing another application from forcibly binding to the same address. If another process has already bound a socket to

	the address provided by the caller, this function will fail.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.

Return Value

If the method succeeds, the return value is a handle to the socket. If the method fails, the return value is `INVALID_SOCKET`. To get extended error information, call **GetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

If the `INET_OPTION_REUSEADDRESS` option is not specified, an error may be returned if a listening socket was recently created for the same local address and port number. By default, once a listening socket is closed there is a period of time that all applications must wait before the address can be reused (this is called the `TIME_WAIT` state). The actual amount of time depends on the operating system and configuration parameters, but is typically two to four minutes. Specifying this option enables an application to immediately re-use a local address and port number that was previously in use. Note that this does not permit more than one server to bind to the same address.

If the `INET_OPTION_EXCLUSIVE` option is specified, the local address and port number cannot be used by another process until the listening socket is closed. This can prevent another application from forcibly binding to the same listening address as your server. This option can be useful in determining whether or not another process is already bound to the address you wish to use, but it may also prevent your server application from restarting immediately, regardless if the `INET_OPTION_REUSEADDRESS` option has also been specified.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the method will fail.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

Unicode: Implemented as Unicode and ANSI versions

See Also

[Accept](#), [DisableEvents](#), [EnableEvents](#), [Reject](#)

CSocketWrench::MatchHostName Method

```
BOOL MatchHostName(  
    LPCTSTR lpszHostName,  
    LPCTSTR lpszHostMask  
    BOOL bResolve  
);
```

The **MatchHostName** method matches a host name against one or more strings that may contain wildcards.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address to match.

lpszHostMask

A null-terminated string which specifies one or more values to match against the host name. The asterisk character can be used to match any number of characters in the host name, and the question mark can be used to match any single character. Multiple values may be specified by separating them with a semicolon.

bResolve

A boolean value which specifies if the host name or IP address should be resolved when matching the host against the mask string. If this parameter is non-zero, two checks against the host mask string will be performed; once for the host name specified and once for its IP address. If this parameter is zero, then the match is made only against the host name string provided.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **MatchHostName** method provides a convenient way for an application to determine if a given host name matches one or more mask strings which may contain wildcard characters. For example, the host name could be "www.microsoft.com" and the host mask string could be "*.microsoft.com". In this example, the method would return a non-zero value indicating the host name matched the mask. However, if the mask string was "*.net" then the method would return zero, indicating that there was no match. Multiple mask values can be combined by separating them with a semicolon; for example, the mask "*.com;*.org" would match any host name in either the .com or .org top-level domains.

If an internationalized domain name (IDN) is specified, it will be converted internally to an ASCII string using Punycode encoding. The host mask will be matched against this encoded version of the host name, not its Unicode version.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

NormalizeHostName Method

```
INT NormalizeHostName(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszNormalized,  
    INT nMaxLength  
);  
  
NormalizeHostName(  
    LPCTSTR lpszHostName,  
    CString& strNormalized  
);
```

The **NormalizeHostName** method returns the canonical form of a host name in the specified buffer.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero-length string.

lpszNormalized

Pointer to the string buffer that will contain the canonical form of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer and must be large enough to store the complete host name. An alternate form of this method will accept a reference to a **CString** object if MFC or ATL is used with the project.

nMaxLength

The maximum number of characters that can be copied to the *lpszNormalized* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the method succeeds, the return value is the number of characters copied into the string buffer. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

The **NormalizeHostName** method will remove all leading and trailing whitespace characters from the host name and fold all upper-case characters to lower-case. If an internationalized domain name (IDN) containing Unicode characters is passed to this method, it will be converted to an ASCII compatible format for domain names.

The *lpszHostName* parameter should only specify a host name or IP address. If you want to support the use of URLs to establish a connection, use the **GetUrlHostName** method which has extended support for extracting the host name and port number specified in a URL.

If the Unicode version of this method is used, the host name will be converted from UTF-16 to UTF-8 and then processed. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended you use the Unicode version.

Although this method performs checks to ensure that the *lpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the **ValidateHostName** method.

It is recommended that you use this method if your application needs to store the host name, and if accepts a host name as user input. It is not necessary to call this method prior to calling the other SocketWrench methods which accept a host name as a parameter. Those methods already normalize the host name and perform checks to ensure it is in the correct format.

If the *lpzHostName* parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this method will return a copy of that IP address in the buffer provided by the caller. This allows the method to be used in cases where a user may input either a host name or IP address. To determine if the IP address has a corresponding host name, use the **GetHostName** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSKV12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetHostAddress](#), [GetHostName](#), [GetUrlHostName](#), [HostNameToUnicode](#), [ValidateHostName](#)

CSocketWrench::Peek Method

```
INT Peek(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **Peek** method reads the specified number of bytes from the socket and copies them into the buffer, but it does not remove the data from the internal socket buffer. The data may be of any type, and is not terminated with a null character.

Parameters

lpBuffer

Pointer to the buffer in which the data will be stored. This argument may be NULL, in which case no data is copied from the socket buffers, however the function will return the number of bytes available to read.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. If the *lpBuffer* parameter is not NULL, this value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes available to read from the socket. A return value of zero indicates that there is no data available to read at that time. If the function fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The **Peek** method returns data that is available to read from the socket, up to the number of bytes specified. The data returned by this method is not removed from the socket buffers. It must be consumed by a subsequent call to the **Read** method. The return value indicates the number of bytes that can be read in a single operation, up to the specified buffer size. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. Using this method in a loop to poll a non-blocking socket may cause the application to become non-responsive. To determine if there is data available to be read, use the **IsReadable** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [Read](#), [ReadLine](#), [RegisterEvent](#), [Write](#), [WriteLine](#)

CSocketWrench::Read Method

```
INT Read(  
    LPBYTE LpBuffer,  
    INT cbBuffer  
);  
  
INT Read(  
    LPBYTE LpBuffer,  
    INT cbBuffer,  
    LPINTERNET_ADDRESS LpAddress,  
    UINT *LpnRemotePort  
);
```

The **Read** method reads the specified number of bytes from the socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

LpBuffer

Pointer to the buffer in which the data will be stored.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

LpAddress

Pointer to an [INTERNET_ADDRESS](#) structure that which will contain the IP address of the remote host that sent the data being read. If this information is not required, the parameter may be specified as NULL.

LpnRemotePort

Pointer to an unsigned integer which will contain the remote port number. If this information is not required, the parameter may be specified as NULL.

Return Value

If the method succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the remote host has closed the connection and there is no more data available to be read. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The **Read** method will read up to the specified number of bytes and store the data in the buffer provided by the caller. If there is no data available to be read at the time this method is invoked, the session thread will block until at least one byte of data becomes available, the timeout period elapses or an error occurs. This method will return if any amount of data is sent by the remote host, and will not block until the entire buffer has been filled. To avoid blocking the current thread, either create an asynchronous socket or use the **IsReadable** method to determine if there is data available to be read prior to calling this function.

The application should never make an assumption about the amount of data that will be available to read. TCP considers all data to be an arbitrary stream of bytes and does not impose any structure on the data itself. For example, if the remote host is sending data to the server in fixed 512 byte blocks of data, it is possible that a single call to the **Read** method will return only a partial block of data, or it

may return multiple blocks combined together. It is the responsibility of the application to buffer and process this data appropriately.

For applications that are built using the Unicode character set, it is important to note that the buffer is an array of bytes, not characters. If the remote host is sending string data to the server, it must be read as a stream of bytes and converted using the **MultiByteToWideChar** function. If the remote host is sending lines of text terminated with a linefeed or carriage return and linefeed pair, the **ReadLine** method will return a line of text at a time and perform this conversion for you.

When **Read** is called and the socket is in non-blocking mode, it is possible that the method will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [Peek](#), [ReadLine](#), [RegisterEvent](#), [Write](#), [WriteLine](#)

CSocketWrench::ReadLine Method

```
BOOL ReadLine(  
    LPTSTR lpszBuffer,  
    LPINT lpnLength  
);  
  
BOOL ReadLine(  
    LPTSTR lpszBuffer,  
    INT nMaxLength  
);  
  
BOOL ReadLine(  
    CString& strBuffer,  
    INT nMaxLength  
);
```

The **ReadLine** method reads up to a line of data from the socket and returns it in a string buffer.

Parameters

lpszBuffer

Pointer to the string buffer that will contain the data when the method returns. The string will be terminated with a null character, and will not contain the end-of-line characters. An alternate form of the method accepts a **CString** argument which will contain the line of text returned by the server.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the method returns, its value will be updated with the actual length of the string.

nMaxLength

An integer value which specifies the maximum length of the buffer.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **ReadLine** method reads data from the socket and copies into a specified string buffer. Unlike the **Read** method which reads arbitrary bytes of data, this method is specifically designed to return a single line of text data in a string. When an end-of-line character sequence is encountered, the method will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters.

There are some limitations when using **ReadLine**. The method should only be used to read text, never binary data. In particular, the method will discard nulls, linefeed and carriage return control characters. The Unicode version of this method will return a Unicode string, however it does not support reading raw Unicode data from the socket. Any data read from the socket is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

This method will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection. If this method is

called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **ReadLine** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **Read** and **ReadLine** method calls can be intermixed, however be aware that **Read** will consume any data that has already been buffered by the **ReadLine** method and this may have unexpected results.

Unlike the **Read** method, it is possible for data to be returned in the string buffer even if the return value is zero. Applications should check the length of the string to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the return value.

Example

```
CString strBuffer;
BOOL bResult;

do
{
    bResult = pSocket->ReadLine(strBuffer);

    if (strBuffer.GetLength() > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = pSocket->GetLastError();

if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The remote host has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[IsReadable](#), [Peek](#), [Read](#), [ReadStream](#), [Write](#), [WriteLine](#), [WriteStream](#)

CSocketWrench::ReadStream Method

```
BOOL ReadStream(  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions,  
    LPBYTE lpMarker,  
    DWORD cbMarker  
);
```

The **ReadStream** method reads the socket data stream and stores the contents in the specified buffer.

Parameters

lpvBuffer

Pointer to the buffer that will contain or reference the data when the method returns. The actual argument depends on the value of the *dwOptions* parameter which specifies how the data stream will be stored.

lpdwLength

A pointer to an unsigned integer value which specifies the maximum length of the buffer and contains the number of bytes read when the method returns. This argument should always point to an initialized value. If the *lpvBuffer* argument specifies a memory buffer, then this argument cannot point to an initialized value of zero; if any other type of stream buffer is used and the initialized value is zero, that indicates that all available data from the socket should be returned until the end-of-stream marker is encountered or the remote host disconnects.

dwOptions

An unsigned integer value which specifies both the stream buffer type and any options to be used when reading the data stream. One of the following stream types may be specified:

Value	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the <i>lpvBuffer</i> parameter. If the argument specifies a pointer to a global memory handle initialized to NULL, then the method will return a handle which references the data; otherwise, the method will consider the parameter a pointer to a block of pre-allocated memory which will contain the stream data when the method returns. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The <i>lpvBuffer</i> argument specifies a pointer to a pre-allocated block of memory which will contain the data read from the socket when the method returns. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the maximum length of the buffer.
INET_STREAM_HGLOBAL	The <i>lpvBuffer</i> argument specifies a pointer to a global memory handle. When the method returns, the handle will reference a block of memory that contains the stream data. The application

	should take care to make sure that the handle passed to the method does not currently reference a valid block of memory; it is recommended that the handle be initialized to NULL prior to calling this method.
INET_STREAM_HANDLE	The <i>lpvBuffer</i> argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read from the socket will be written to this handle using the WriteFile function.
INET_STREAM_SOCKET	The <i>lpvBuffer</i> argument specifies a socket handle. The data read from the socket specified by the <i>hSocket</i> argument will be written to this socket. The socket handle passed to this method must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either create another instance of the class and attach the socket using the AttachHandle method or use the INET_STREAM_HANDLE stream buffer type instead.
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being returned as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the source data stream; if the application is using a pre-allocated memory buffer, this must be considered before calling this method.

lpMarker

A pointer to an array of bytes which marks the end of the data stream. When this byte sequence is encountered by the method, it will stop reading and return to the caller. The buffer will contain all of the data read from the socket up to and including the end-of-stream marker. If this argument is NULL, then the method will continue to read from the socket until the maximum buffer size is reached, the remote host closes its socket or an error is encountered.

cbMarker

An unsigned integer value which specifies the length of the end-of-stream marker in bytes. If the *lpMarker* parameter is NULL, then this value must be zero.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To

get extended error information, call **GetLastError**.

Remarks

The **ReadStream** method enables an application to read an arbitrarily large stream of data and store it in memory, write it to a file or even another socket. Unlike the **Read** method, which will return immediately when any amount of data has been read, **ReadStream** will only return when the buffer is full as specified by the *lpdwLength* parameter, the logical end-of-stream marker has been read, the socket closed by the remote host or when an error occurs.

This method will force the thread to block until the operation completes. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **ReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for data to be returned in the buffer even if the method returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the method is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the method return value.

Because **ReadStream** can potentially cause the application to block for long periods of time as the data stream is being read, the method will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **RegisterEvent** method, and can obtain information about the current operation by calling the **GetStreamInfo** method.

Example

```
HGLOBAL hgb1Buffer = NULL; // Return data in a global memory buffer
DWORD cbBuffer = 102400;   // Read up to 100K bytes
BOOL bResult;

bResult = pSocket->ReadStream(&hgb1Buffer, &cbBuffer,
                              INET_STREAM_HGLOBAL | INET_STREAM_CONVERT);

if (bResult && cbBuffer > 0)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgb1Buffer);

    // Use data in the stream buffer

    GlobalUnlock(hgb1Buffer);
    GlobalFree(hgb1Buffer);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

CSocketWrench::RegisterEvent Method

```
INT RegisterEvent(  
    UINT nEventId,  
    INETEVENTPROC lpEventProc,  
    DWORD_PTR dwParam  
);
```

The **RegisterEvent** method registers an event handler for the specified event.

Parameters

nEventId

An unsigned integer which specifies which event should be registered with the specified callback function. One of the following values may be used:

Value	Description
INET_EVENT_CONNECT	The connection to the remote host has completed.
INET_EVENT_DISCONNECT	The remote host has closed the connection to the client. The client should read any remaining data and disconnect.
INET_EVENT_READ	A network event which indicates data is available to read. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	A network event which indicates the application can send data to the remote host. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The application may attempt to retry the operation, or may disconnect from the remote host and report an error to the user.
INET_EVENT_CANCEL	The current operation has been canceled. Under most circumstances the client should disconnect from the server and re-connect if needed. After an operation has been canceled, the server may abort the connection or refuse to accept further commands from the client.

lpEventProc

Specifies the procedure-instance address of the application defined callback function. For more information about the callback function, see the description of the **InetEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR.

To get extended error information, call **GetLastError**.

Remarks

The **RegisterEvent** method associates a callback function with a specific event. The event handler is an **InetEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

The *dwParam* parameter is commonly used to identify the class instance which is associated with the event that has occurred. Applications will cast the **this** pointer to a **DWORD_PTR** value when calling this function, and then the event handler will cast it back to a pointer to the class instance. This gives the handler access to the class member variables and methods.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[DisableEvents](#), [EnableEvents](#), [FreezeEvents](#), [InetEventProc](#)

CSocketWrench::Reject Method

BOOL Reject();

The **Reject** method is used to reject a client connection request.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **Reject** method rejects a pending client connection and the remote host will see this as the connection being aborted. If there are no pending client connections at the time, this method will immediately return with an error indicating that the operation would cause the thread to block.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[Accept](#), [Listen](#)

CSocketWrench::SetHostFile Method

```
INT SetHostFile(  
    LPCTSTR lpszFileName  
);
```

The **SetHostFile** method specifies the name of an alternate file to use when resolving hostnames and IP addresses. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **GetHostAddress** and **GetHostNames** method. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Parameters

lpszFileName

Pointer to a string that specifies the name of the file. If the parameter is NULL, then the current host file is cleared from the cache and only the default host file will be used to resolve hostnames and addresses.

Return Value

If the method succeeds, the return value is the number of entries in the host file. A return value of INET_ERROR indicates failure. To get extended error information, call **GetLastError**.

Remarks

This method loads the file into memory allocated for the current thread. If the contents of the file have changed after the method has been called, those changes will not be reflected when resolving hostnames or addresses. To reload the host file from disk, call this method again with the same file name. To remove the alternate host file from memory, specify a NULL pointer as the parameter.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

To determine if an alternate host file has been specified, use the **GetHostFile** method. A return value of zero indicates that no alternate host file has been cached for the current thread.

A system may have a default host file, which is used to resolve hostnames before performing a nameserver lookup. To determine the name of this file, use the **GetDefaultHostFile** method. It is not necessary to specify this default host file, since it is always used to resolve host names and addresses.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csowskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetDefaultHostFile](#), [GetHostAddress](#), [GetHostFile](#), [GetHostName](#)

CSocketWrench::SetLastError Method

```
VOID SetLastError(  
    DWORD dwErrorCode  
);
```

The **SetLastError** method sets the last error code for the current thread. This method is typically used to clear the last error by specifying a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last error code for the current thread.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most methods will set the last error code value when they fail; a few methods set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those methods which call **SetLastError** when they succeed are noted on the method reference page.

Applications can retrieve the value saved by this method by using the **GetLastError** method. The use of **GetLastError** is optional; an application can call the method to determine the specific reason for a method failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetErrorString](#), [GetLastError](#), [ShowError](#)

CSocketWrench::SetOption Method

```
INT SetOption(  
    DWORD dwOption,  
    BOOL bEnabled  
);
```

The **SetOption** method is used to enable or disable a specific socket option.

Parameters

dwOption

An unsigned integer used to specify one of the socket options. These options cannot be combined. The following values are recognized:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_KEEPAIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.

bEnabled

A boolean flag. If the flag is set to a non-zero value, the option is enabled. Otherwise the socket option is disabled.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

It is not recommend that you disable the Nagle algorithm by specifying the INET_OPTION_NODELAY flag unless it is absolutely required. Doing so can have a significant, negative impact on the performance of the application and network.

If if the INET_OPTION_KEEPAIVE option is enabled, keep-alive packets will start being generated five seconds after the socket has become idle with no data being sent or received. Enabling this option can be used by applications to detect when a physical network connection has been lost. However, it is recommended that most applications query the remote host directly to determine if the connection is still active. This is typically accomplished by sending specific commands to the server to query its status, or checking the elapsed time since the last response from the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[Connect](#), [GetOption](#)

CSocketWrench::SetTimeout Method

```
INT SetTimeout(  
    UINT nTimeout  
);
```

The **SetTimeout** method sets the number of seconds the client will wait for a response from the remote host. Once the specified number of seconds has elapsed, the method will fail and return to the caller.

Parameters

nTimeout

The number of seconds to wait for a blocking operation to complete.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is `INET_ERROR`. To get extended error information, call **GetLastError**.

Remarks

The timeout value is only used with blocking client connections.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[Connect](#), [GetTimeout](#), [IsReadable](#), [IsWritable](#), [Read](#), [Write](#)

CSocketWrench::ShowError Method

```
INT ShowError(  
    LPCTSTR lpszAppTitle,  
    UINT uType,  
    DWORD dwErrorCode  
);
```

The **ShowError** method displays a message box which describes the specified error.

Parameters

lpszAppTitle

A null-terminated string which specifies the title of the message box that is displayed. If this argument is NULL or omitted, then the default title of "Error" will be displayed.

uType

An unsigned integer which specifies the type of message box that will be displayed. This is the same value that is used by the **MessageBox** method in the Windows API. If a value of zero is specified, then a message box with a single OK button will be displayed. Refer to that method for a complete list of options.

dwErrorCode

Specifies the error code that will be used when displaying the message box. If this argument is zero, then the last error that occurred in the current thread will be displayed.

Return Value

If the method is successful, the return value will be the return value from the **MessageBox** function. If the method fails, it will return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetErrorString](#), [GetLastError](#), [SetLastError](#)

CSocketWrench::Shutdown Method

```
INT Shutdown(  
    DWORD dwOption  
);
```

The **Shutdown** method is used to disable reception or transmission of data, or both.

Parameters

dwOption

An unsigned integer used to specify one of the shutdown options. These options cannot be combined. The following values are recognized:

Value	Description
INET_SHUTDOWN_READ	Disable reception of data.
INET_SHUTDOWN_WRITE	Disable transmission of data.
INET_SHUTDOWN_BOTH	Disable both reception and transmission of data.

Return Value

If the method succeeds, the return value is zero. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

This method is rarely needed. It is provided as an interface to the Windows Sockets **shutdown** method.

In some asynchronous applications, it may be desirable for a client to inform the server that no further communication is wanted, while allowing the client to read any residual data that may reside in internal buffers on the client side. **Shutdown** accomplishes this because the socket handle is still valid after it has been called, although some or all communication with the remote host has ceased.

Requirements

- Minimum Desktop Platform:** Windows 7 Service Pack 1
- Minimum Server Platform:** Windows Server 2008 R2 Service Pack 1
- Header File:** cswsock12.h
- Import Library:** cswskv12.lib

See Also

[Abort](#), [Connect](#), [Disconnect](#)

CSocketWrench::StoreStream Method

```
BOOL StoreStream(  
    LPCTSTR lpszFileName,  
    DWORD dwLength,  
    LPDWORD lpdwCopied  
    DWORD dwOffset,  
    DWORD dwOptions  
);
```

The **StoreStream** method reads the socket data stream and stores the contents in the specified file.

Parameters

- lpszFileName*
- Pointer to a string which specifies the name of the file to create or overwrite.
- dwLength*
- An unsigned integer which specifies the maximum number of bytes to read from the socket and write to the file. If this value is zero, then the method will continue to read data from the socket until the remote host disconnects or an error occurs.
- lpdwCopied*
- A pointer to an unsigned integer value which will contain the number of bytes written to the file when the method returns.
- dwOffset*
- An unsigned integer which specifies the byte offset into the file where the method will start storing data read from the socket. Note that all data after this offset will be truncated. A value of zero specifies that the file should be completely overwritten if it already exists.
- dwOptions*
- An unsigned integer value which specifies one or more options. Programs can use a bitwise operator to combine any of the following values:

Value	Description
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data written to the file to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being written as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the

source data stream. If the <i>dwOffset</i> parameter has a value of zero, the Unicode byte order mark (BOM) will be written to the beginning of the file.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **StoreStream** method enables an application to read an arbitrarily large stream of data and store it in a file. This method is essentially a simplified version of the **ReadStream** method, designed specifically to be used with files rather than memory buffers or handles.

This method will force the thread to block until the operation completes. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **StoreStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

Because **StoreStream** can potentially cause the application to block for long periods of time as the data stream is being read, the method will periodically generate `INET_EVENT_PROGRESS` events. An application can register an event handler using the **RegisterEvent** method, and can obtain information about the current operation by calling the **GetStreamInfo** method.

Example

```
DWORD dwCopied;
BOOL bResult;

bResult = pSocket->StoreStream(lpszFileName, 0, &dwCopied, 0,
                               INET_STREAM_CONVERT);

if (bResult && dwCopied > 0)
{
    // The data has been written to the file
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

Unicode: Implemented as Unicode and ANSI versions

See Also

[GetStreamInfo](#), [Read](#), [ReadLine](#), [ReadStream](#), [Write](#), [WriteLine](#), [WriteStream](#)

CSocketWrench::ValidateCertificate Method

```
BOOL ValidateCertificate(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertPassword,  
    LPCTSTR lpszCertName  
);
```

The **ValidateCertificate** method determines if the specified security certificate is installed on the local system.

Parameters

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the personal certificate store will be used as the default. This parameter may also specify the name of a certificate file in PKCS #12 (PFX) format.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertPassword

A null-terminated string which specifies the password associated with a certificate file. This parameter is only used if the *lpszCertStore* parameter specifies a certificate file, otherwise it is ignored. If the certificate file is not protected with a password, this parameter should be a NULL pointer or empty string.

lpszCertName

A null-terminated string which specifies the name of the certificate to validate. The method will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the method will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the method will return an error indicating that the certificate could not be found.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

If you are checking the validity of a certificate installed in the local certificate store, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user.

It is possible to validate a certificate file rather than one stored in the local certificate store. The *lpzCertStore* member should specify the name of a file in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate file was created, it must be provided in with the *lpzCertPassword* parameter or this method will be unable to access the certificate.

This method can only validate certificate files in PFX format and cannot be used to validate a certificate file in another format, such as PEM (base64 encoded) or DER (binary).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[CreateSecurityCredentials](#), [DeleteSecurityCredentials](#)

ValidateHostName Method

```
BOOL ValidateHostName(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

```
BOOL ValidateHostName(  
    LPCTSTR lpszHostName,  
    CString& strAddress  
);
```

The **ValidateHostName** method determines if the specified host name is valid and returns its IP address.

Parameters

lpszHostName

A null-terminated string which specifies the host name. The method will fail If this parameter is NULL or an empty string.

lpszAddress

A null-terminated string buffer which will contain the IP address of the host. If specified, this string must be large enough to store the complete IP address, including the terminating null character. If this parameter is NULL or the *nMaxLength* parameter is zero, it will be ignored and the IP address will not be returned. An alternate version of this method accepts a reference to a **CString** object if MFC or ATL is used with the project.

nMaxLength

An integer value that specifies the maximum number of characters which can be copied into the *lpszAddress* string buffer. The buffer must be large enough to store the complete address. Because this method can return either an IPv4 or IPv6 address, it is recommended the minimum length for the buffer to be 46 characters. If this parameter is zero, the *lpszAddress* parameter will be ignored.

Return Value

If the method succeeds, the host name is valid and the return value will be non-zero. If the method fails, the host name could not be resolved to an IP address and the return value will be zero. To get extended error information, call the **GetLastError** method.

Remarks

The **ValidateHostName** method provides a convenient way to determine if a host name is valid by attempting to resolve the name into an IP address. It is similar to calling the **NormalizeHostName** method to obtain the canonical form of the host name, calling **GetAddress** to obtain the IP address and then calling **FormatAddress** to return the string representation of the host's IP address.

If the Unicode version of this method is used, any non-ASCII characters in the host name will be automatically encoded into a compatible format and then resolved to an IP address. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended you use the Unicode version.

The *lpszHostName* parameter can only specify a host name or IP address and cannot be a URL. If you want your application to support providing a URL in addition to a host name, use the **GetUrlHostName** method to extract the host name from the URL. You can then provide the host

name to this method to obtain its IP address.

If the *lpzHostName* parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this method will return a copy of that IP address in the buffer provided by the caller. This allows the method to be used in cases where a user may input either a host name or IP address. To determine if the IP address has a corresponding host name, use the **GetHostName** method.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[FormatAddress](#), [GetAddress](#), [GetHostAddress](#), [GetHostName](#), [GetUrlHostName](#), [NormalizeHostName](#)

CSocketWrench::Write Method

```
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);  
  
INT Write(  
    LPBYTE lpBuffer,  
    INT cbBuffer,  
    LPINTERNET_ADDRESS lpAddress,  
    UINT nRemotePort  
);
```

The **Write** method sends the specified number of bytes to the remote host.

Parameters

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the remote host.

cbBuffer

The number of bytes to send from the specified buffer. This value must be greater than zero.

lpAddress

Pointer to an [INTERNET_ADDRESS](#) structure that specifies the address of the remote host that is to receive the data being written. For TCP stream sockets, this parameter must always be NULL or specify the same address that was used to establish the connection. For UDP datagram sockets, this may specify any valid IP address.

nRemotePort

The port number of the remote host that is to receive the data being written. For TCP stream sockets, this value must always be zero, or specify the same port number that was used to establish the connection. For UDP datagram sockets, this may specify any valid port number.

Return Value

If the method succeeds, the return value is the number of bytes actually written. If the method fails, the return value is INET_ERROR. To get extended error information, call **GetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the client application to send the remaining data at some later point. For non-blocking sockets, the client must wait for the next asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[EnableEvents](#), [IsBlocking](#), [IsReadable](#), [IsWritable](#), [Read](#), [ReadLine](#), [RegisterEvent](#), [WriteLine](#)

CSocketWrench::WriteLine Method

```
BOOL WriteLine(  
    LPCTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

The **WriteLine** function sends a line of text to the remote host, terminated by a carriage-return and linefeed.

Parameters

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the remote host. All characters up to, but not including, the terminating null character will be written to the socket. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then a only a carriage-return and linefeed are written to the socket.

lpnLength

A pointer to an integer value which will contain the number of characters written to the socket, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **WriteLine** method writes a line of text to the remote host and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **Write** method which writes arbitrary bytes of data to the socket, this method is specifically designed to write a single line of text data from a null-terminated string.

If the *lpszBuffer* string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the value returned in the *lpnLength* parameter will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

There are some limitations when using **WriteLine**. This method should only be used to send text, never binary data. In particular, it will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this method will accept a Unicode string, however it does not support writing raw Unicode data to the socket. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written to the socket as a stream of bytes.

This method will force the thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, send the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **WriteLine** is blocked sending data to the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application

needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **Write** and **WriteLine** methods can be safely intermixed.

Unlike the **Write** function, it is possible for data to have been written to the socket if the return value is zero. For example, if a timeout occurs while the method is waiting to send more data to the remote host, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the *lpnLength* argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[IsWritable](#), [Read](#), [ReadLine](#), [ReadStream](#), [Write](#), [WriteStream](#)

CSocketWrench::WriteStream Method

```
BOOL WriteStream(  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

The **WriteStream** method writes data from the stream buffer to the specified socket.

Parameters

lpvBuffer

Pointer to the buffer that contains or references the data to be written to the socket. The actual argument depends on the value of the *dwOptions* parameter which specifies how the data stream will be accessed.

lpdwLength

A pointer to an unsigned integer value which specifies the size of the buffer and contains the number of bytes written when the method returns. This argument should should always point to an initialized value. If the *lpvBuffer* argument specifies a memory buffer or global memory handle, then this argument cannot point to an initialized value of zero.

dwOptions

An unsigned integer value which specifies the stream buffer type to be used when writing the data stream to the socket. One of the following stream types may be specified:

Value	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the <i>lpvBuffer</i> parameter. If the argument specifies a global memory handle, then the method will write the data referenced by that handle; otherwise, the method will consider the parameter a pointer to a block of memory which contains data to be written. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The <i>lpvBuffer</i> argument specifies a pointer to a block of memory which contains the data to be written to the socket. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HGLOBAL	The <i>lpvBuffer</i> argument specifies a global memory handle that references the data to be written to the socket. The handle must have been created by a call to the GlobalAlloc or GlobalReAlloc function. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HANDLE	The <i>lpvBuffer</i> argument specifies a Windows handle to

	an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read using the ReadFile function with this handle will be written to the socket.
INET_STREAM_SOCKET	The <i>lpvBuffer</i> argument specifies a socket handle. The data read from the socket specified by this handle will be written to the socket specified by the <i>hSocket</i> parameter. The socket handle passed to this method must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either create another instance of the class and attach the socket using the AttachHandle method or use the INET_STREAM_HANDLE stream buffer type instead.

Return Value

If the method succeeds, the return value is non-zero. If the method fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **WriteStream** method enables an application to write an arbitrarily large stream of data from memory or a file to the specified socket. Unlike the **Write** method, which may not write all of the data in a single method call, **WriteStream** will only return when all of the data has been written or an error occurs.

This method will force the thread to block until the operation completes. If this method is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, write the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **WriteStream** is blocked sending data to the remote host, an error will occur. It is recommended that this method only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for some data to have been written even if the method returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was sent. For example, if a timeout occurs while the method is waiting to write more data, it will return zero; however, some data may have already been written to the socket prior to the error condition.

Because **WriteStream** can potentially cause the application to block for long periods of time as the data stream is being written, the method will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **RegisterEvent** method, and can obtain information about the current operation by calling the **GetStreamInfo** method.

Example

```
CFile *pFile = new CFile();
DWORD dwLength = 0;

if (!pFile->Open(strFileName, CFile::modeRead | CFile::shareDenyWrite))
    return;
```

```
dwLength = pFile->GetLength();

if (dwLength > 0)
{
    BOOL bResult = pSocket->WriteStream(
        pFile->m_hFile,
        &dwLength,
        INET_STREAM_HANDLE);
}

delete pFile;
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[GetStreamInfo](#), [Read](#), [ReadLine](#), [ReadStream](#), [StoreStream](#), [Write](#), [WriteLine](#)

SocketWrench Data Structures

- INETSTREAMINFO
- INTERNET_ADDRESS
- SECURITYCREDENTIALS
- SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

INETSTREAMINFO

The INETSTREAMINFO structure contains information about the data stream being currently read or written.

```
typedef struct _INETSTREAMINFO
{
    DWORD    dwStreamThread;
    DWORD    dwStreamSize;
    DWORD    dwStreamCopied;
    DWORD    dwStreamMode;
    DWORD    dwStreamError;
    DWORD    dwBytesPerSecond;
    DWORD    dwTimeElapsed;
    DWORD    dwTimeEstimated;
} INETSTREAMINFO, *LPINETSTREAMINFO;
```

Members

dwStreamThread

Specifies the numeric ID for the thread that created the socket.

dwStreamSize

The maximum number of bytes that will be read or written. This is the same value as the buffer length specified by the caller, and may be zero which indicates that no maximum size was specified. Note that if this value is zero, the application will be unable to calculate a completion percentage or estimate the amount of time for the operation to complete.

dwStreamCopied

The total number of bytes that have been copied to or from the stream buffer.

dwStreamMode

A numeric value which specifies the stream operation that is current being performed. It may be one of the following values:

Value	Description
INET_STREAM_READ	Data is being read from the socket and stored in the specified stream buffer.
INET_STREAM_WRITE	Data is being written from the specified stream buffer to the socket.

dwStreamError

The last error that occurred when reading or writing the data stream. If no error has occurred, this value will be zero.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the operation is completed. This is based on the average number of bytes transferred per second and requires that a maximum stream buffer size be specified by the caller.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

See Also

[ReadStream](#), [StoreStream](#), [WriteStream](#)

INTERNET_ADDRESS

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
    INT    ipFamily;
    BYTE   ipNumber[16];
} INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the <i>ipNumber</i> array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The **INTERNET_ADDRESS** structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the *ipNumber* array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the *ipFamily* member first to make sure that it is actually an IPv4 address.

Requirements

- Minimum Desktop Platform:** Windows 7 Service Pack 1
- Minimum Server Platform:** Windows Server 2008 R2 Service Pack 1
- Header File:** cswsock12.h

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpzCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpzPassword*** member or the library will be unable to access the certificate.

Note that the ***lpzUserName*** and ***lpzPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cstools12.h

Unicode: Implemented as Unicode and ANSI versions

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME    stCertIssued;
    SYSTEMTIME    stCertExpires;
    LPCTSTR       lpszCertIssuer;
    LPCTSTR       lpszCertSubject;
    LPCTSTR       lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KED	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpzCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpzFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpzCertIssuer*** and ***lpzCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: ctools12.h

Unicode: Implemented as Unicode and ANSI versions

SocketWrench Class Error Codes

Value	Constant	Description
0x80042711	ST_ERROR_NOT_HANDLE_OWNER	Handle not owned by the current thread
0x80042712	ST_ERROR_FILE_NOT_FOUND	The specified file or directory does not exist
0x80042713	ST_ERROR_FILE_NOT_CREATED	The specified file could not be created
0x80042714	ST_ERROR_OPERATION_CANCELED	The blocking operation has been canceled
0x80042715	ST_ERROR_INVALID_FILE_TYPE	The specified file is a block or character device, not a regular file
0x80042716	ST_ERROR_INVALID_DEVICE	The specified device or address does not exist
0x80042717	ST_ERROR_TOO_MANY_PARAMETERS	The maximum number of method parameters has been exceeded
0x80042718	ST_ERROR_INVALID_FILE_NAME	The specified file name contains invalid characters or is too long
0x80042719	ST_ERROR_INVALID_FILE_HANDLE	Invalid file handle passed to method
0x8004271A	ST_ERROR_FILE_READ_FAILED	Unable to read data from the specified file
0x8004271B	ST_ERROR_FILE_WRITE_FAILED	Unable to write data to the specified file
0x8004271C	ST_ERROR_OUT_OF_MEMORY	Out of memory
0x8004271D	ST_ERROR_ACCESS_DENIED	Access denied
0x8004271E	ST_ERROR_INVALID_PARAMETER	Invalid argument passed to method
0x8004271F	ST_ERROR_CLIPBOARD_UNAVAILABLE	The system clipboard is currently unavailable
0x80042720	ST_ERROR_CLIPBOARD_EMPTY	The system clipboard is empty or does not contain any text data
0x80042721	ST_ERROR_FILE_EMPTY	The specified file does not contain any data
0x80042722	ST_ERROR_FILE_EXISTS	The specified file already exists
0x80042723	ST_ERROR_END_OF_FILE	End of file
0x80042724	ST_ERROR_DEVICE_NOT_FOUND	The specified device could not be found
0x80042725	ST_ERROR_DIRECTORY_NOT_FOUND	The specified directory could not be found
0x80042726	ST_ERROR_INVALID_BUFFER	Invalid memory address passed to method
0x80042728	ST_ERROR_NO_HANDLES	No more handles available to this process
0x80042733	ST_ERROR_OPERATION_WOULD_BLOCK	The specified operation would block the current thread
0x80042734	ST_ERROR_OPERATION_IN_PROGRESS	A blocking operation is currently in progress
0x80042735	ST_ERROR_ALREADY_IN_PROGRESS	The specified operation is already in progress
0x80042736	ST_ERROR_INVALID_HANDLE	Invalid handle passed to method
0x80042737	ST_ERROR_INVALID_ADDRESS	Invalid network address specified

0x80042738	ST_ERROR_INVALID_SIZE	Datagram is too large to fit in specified buffer
0x80042739	ST_ERROR_INVALID_PROTOCOL	Invalid network protocol specified
0x8004273A	ST_ERROR_PROTOCOL_NOT_AVAILABLE	The specified network protocol is not available
0x8004273B	ST_ERROR_PROTOCOL_NOT_SUPPORTED	The specified protocol is not supported
0x8004273C	ST_ERROR_SOCKET_NOT_SUPPORTED	The specified socket type is not supported
0x8004273D	ST_ERROR_INVALID_OPTION	The specified option is invalid
0x8004273E	ST_ERROR_PROTOCOL_FAMILY	Specified protocol family is not supported
0x8004273F	ST_ERROR_PROTOCOL_ADDRESS	The specified address is invalid for this protocol family
0x80042740	ST_ERROR_ADDRESS_IN_USE	The specified address is in use by another process
0x80042741	ST_ERROR_ADDRESS_UNAVAILABLE	The specified address cannot be assigned
0x80042742	ST_ERROR_NETWORK_UNAVAILABLE	The networking subsystem is unavailable
0x80042743	ST_ERROR_NETWORK_UNREACHABLE	The specified network is unreachable
0x80042744	ST_ERROR_NETWORK_RESET	Network dropped connection on remote reset
0x80042745	ST_ERROR_CONNECTION_ABORTED	Connection was aborted due to timeout or other failure
0x80042746	ST_ERROR_CONNECTION_RESET	Connection was reset by remote network
0x80042747	ST_ERROR_OUT_OF_BUFFERS	No buffer space is available
0x80042748	ST_ERROR_ALREADY_CONNECTED	Connection already established with remote host
0x80042749	ST_ERROR_NOT_CONNECTED	No connection established with remote host
0x8004274A	ST_ERROR_CONNECTION_SHUTDOWN	Unable to send or receive data after connection shutdown
0x8004274C	ST_ERROR_OPERATION_TIMEOUT	The specified operation has timed out
0x8004274D	ST_ERROR_CONNECTION_REFUSED	The connection has been refused by the remote host
0x80042750	ST_ERROR_HOST_UNAVAILABLE	The specified host is unavailable
0x80042751	ST_ERROR_HOST_UNREACHABLE	Remote host is unreachable
0x80042753	ST_ERROR_TOO_MANY_PROCESSES	Too many processes are using the networking subsystem
0x8004276B	ST_ERROR_NETWORK_NOT_READY	Network subsystem is not ready for communication
0x8004276C	ST_ERROR_INVALID_VERSION	This version of the operating system is not supported
0x8004276D	ST_ERROR_NETWORK_NOT_INITIALIZED	The networking subsystem has not been initialized
0x80042775	ST_ERROR_REMOTE_SHUTDOWN	The remote host has initiated a graceful shutdown sequence

0x80042AF9	ST_ERROR_INVALID_HOSTNAME	The specified hostname is invalid or could not be resolved
0x80042AFA	ST_ERROR_HOSTNAME_NOT_FOUND	The specified hostname could not be found
0x80042AFB	ST_ERROR_HOSTNAME_REFUSED	Unable to resolve hostname, request refused
0x80042AFC	ST_ERROR_HOSTNAME_NOT_RESOLVED	Unable to resolve hostname, no address for specified host
0x80042EE1	ST_ERROR_INVALID_LICENSE	The license for this product is invalid
0x80042EE2	ST_ERROR_PRODUCT_NOT_LICENSED	This product is not licensed to perform this operation
0x80042EE3	ST_ERROR_NOT_IMPLEMENTED	This method has not been implemented on this platform
0x80042EE4	ST_ERROR_UNKNOWN_LOCALHOST	Unable to determine local host name
0x80042EE5	ST_ERROR_INVALID_HOSTADDRESS	Invalid host address specified
0x80042EE6	ST_ERROR_INVALID_SERVICE_PORT	Invalid service port number specified
0x80042EE7	ST_ERROR_INVALID_SERVICE_NAME	Invalid or unknown service name specified
0x80042EE8	ST_ERROR_INVALID_EVENTID	Invalid event identifier specified
0x80042EE9	ST_ERROR_OPERATION_NOT_BLOCKING	No blocking operation in progress on this socket
0x80042F45	ST_ERROR_SECURITY_NOT_INITIALIZED	Unable to initialize security interface for this process
0x80042F46	ST_ERROR_SECURITY_CONTEXT	Unable to establish security context for this session
0x80042F47	ST_ERROR_SECURITY_CREDENTIALS	Unable to open client certificate store or establish client credentials
0x80042F48	ST_ERROR_SECURITY_CERTIFICATE	Unable to validate the certificate chain for this session
0x80042F49	ST_ERROR_SECURITY_DECRYPTION	Unable to decrypt data stream
0x80042F4A	ST_ERROR_SECURITY_ENCRYPTION	Unable to encrypt data stream
0x80043031	ST_ERROR_MAXIMUM_CONNECTIONS	The maximum number of client connections exceeded
0x80043032	ST_ERROR_THREAD_CREATION_FAILED	Unable to create a new thread for the current process
0x80043033	ST_ERROR_INVALID_THREAD_HANDLE	The specified thread handle is no longer valid
0x80043034	ST_ERROR_THREAD_TERMINATED	The specified thread has been terminated
0x80043035	ST_ERROR_THREAD_DEADLOCK	The operation would result in the current thread becoming deadlocked
0x80043036	ST_ERROR_INVALID_CLIENT_MONIKER	The specified moniker is not associated with any client session
0x80043037	ST_ERROR_CLIENT_MONIKER_EXISTS	The specified moniker has been assigned to another client session

0x80043038	ST_ERROR_SERVER_INACTIVE	The specified server is not listening for client connections
0x80043039	ST_ERROR_SERVER_SUSPENDED	The specified server is suspended and not accepting client connections

SocketWrench Library Overview

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

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

Data Type Representations

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

Description	Size	Range	C / C++	Visual Basic	Visual Basic.NET
Byte	1 byte	0 to 255	BYTE	Byte	Byte
Boolean	4 bytes	0 is False, 1 is True	BOOL	Long	Integer
Handle	4	0 to 4,294,967,295	HPACKAGE	Long	Integer

	bytes				
Integer	4 bytes	-2,147,483,648 to 2,147,483,647	int	Long	Integer
Integer	4 bytes	0 to 4,294,967,295	UINT	Long	Integer
Short Integer	2 bytes	-32,768 to 32,767	SHORT	Integer	Short
Short Integer	2 bytes	0 to 65,535	WORD	Integer	Short
Long Integer	4 bytes	-2,147,483,648 to 2,147,483,647	LONG	Long	Integer
Long Integer	4 bytes	0 to 4,294,967,295	DWORD	Long	Integer

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

Handles

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

In SocketWrench, handles are used to refer to sockets. The socket handle, defined as an unsigned integer type called SOCKET, is returned by those functions which create a new outbound connection, or listen for inbound connections. During the time that this connection exists, the socket handle is used to refer to connection. When the application closes the connection, the handle is destroyed.

The value of an SocketWrench handle varies between processes. The only constant is the value represented by INVALID_SOCKET, which indicates that a given handle does not represent an open socket.

Boolean Data

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

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

```
Dim bConnected As Long
```

```
bConnected = InetIsConnected(hSocket)
```



```

If bResult = True Then
    ' The socket is connected to a remote host
Else
    ' The socket is not connected, or the socket handle
    ' may be invalid; call InetGetLastError to check the
    ' error code
End If

```

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

```

Dim bConnected As Long

bConnected = InetIsConnected(hSocket)

If bResult <> 0 Then
    ' The socket is connected to a remote host
Else
    ' The socket is not connected, or the socket handle
    ' may be invalid; call InetGetLastError to check the
    ' error code
End If

```

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

String Data

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

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

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

```

Dim strLocalName As String
Dim nLength As Long

nLength = InetGetLocalName(strLocalName, 255)

```

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

```

Dim strLocalName As String * 256
Dim nLength As Long

```

```
nLength = InetGetLocalName(strLocalName, 255)
```

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

```
Dim strLocalName As String  
Dim nLength As Long
```

```
strLocalName = String(256, 0)  
nLength = InetGetLocalName(strLocalName, 255)
```

One final consideration is that string data that is returned by the function will be null-terminated. For those languages that do not terminate strings with null characters, it may be necessary to remove the trailing null character. The complete example would be written in Visual Basic as:

```
Dim strLocalName As String  
Dim nLength As Long
```

```
strLocalName = String(256, 0)  
nLength = InetGetLocalName(strLocalName, 255)
```

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

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

Unicode Support

Unicode is a multi-language character set designed to encompass virtually all of the characters used with computers today. Unicode characters are represented by a 16-bit value, and differ from other character sets in two important ways. First, unlike the traditional single-byte (ANSI) character sets, Unicode is capable of representing significantly more characters in a variety of languages. Second, unlike multi-byte character sets (where some characters may be one byte in length, while others may be two bytes), the characters are fixed-width, which makes them easier to work with. Windows support for Unicode varies according to the platform. Unicode is fully supported under Windows NT and later versions of the operating system. However, support is limited under Windows 95/98.

The SocketWrench library supports both the ANSI and Unicode character sets by providing two versions of each function that either expects a string as an argument (including those functions which pass structures that contain strings) or returns the address of a string.

The version of the function that expects a single-byte character set has a suffix of "A" (ANSI), while the function which expects the Unicode character set has a suffix of "W" (wide). Note that no suffix is used with those functions which only expect numeric parameters and return numeric values.

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

Some SocketWrench functions expect byte arrays, not character strings. This can create problems when reading and writing Unicode string data. For example, consider the **InetRead** and **InetWrite** functions which

are used to read and write data on the socket. Because character strings and byte arrays are essentially identical when using 8-bit ANSI character sets, a C/C++ programmer may try to write code such as this:

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

This would work as expected until the project is changed to use Unicode. The problem is the string is no longer an array of bytes, but is now an array of 16-bit (short) integers. The string must be converted from Unicode to a byte array before passing it to the **InetWrite** function. To do this, the **WideCharToMultiByte** function can be used as follows:

```
LPTSTR lpszData;  
LPBYTE lpBuffer;  
int cchData, nResult;  
  
lpszData = _T("This is a test, this is only a test");  
cchData = lstrlen(lpszData);  
  
#ifdef UNICODE  
if ((lpBuffer = (LPBYTE)_alloca((cchData + 1) * 4)) == NULL)  
{  
    // Unable to allocate memory  
    return;  
}  
  
WideCharToMultiByte(CP_UTF8, 0,  
                    (LPCWSTR)lpszData, cchData,  
                    (LPSTR)lpBuffer, ((cchData + 1) * 4),  
                    NULL, NULL);  
  
#else  
lpBuffer = (LPBYTE)lpszData;  
#endif  
  
nResult = InetWrite(hSocket, lpBuffer, cchData);
```

Note that the type of characters that are being converted may also present a problem to the developer. In this example, the string is easily converted because it is composed only of characters that are part of the basic ASCII character set. However, when converting a string that contains international characters (such as accented vowels) the conversion may result in unprintable characters. For additional information, check your programming language's technical reference for issues with regards to localization and the use of Unicode.

SocketWrench Library and Visual Basic

SocketWrench includes both an ActiveX control and a standard Windows dynamic link library, and in most cases those developers who are programming in Visual Basic will benefit from using the control. However, there are those situations in which a Visual Basic programmer may prefer to use the library over the control.

SocketWrench does not come with Visual Basic modules which provide function declarations and constants. Instead, this information is provided as a type library, which is part of the library itself. To use the SocketWrench library in your Visual Basic programs, you need to create a reference to it in your project. To do this, follow these steps:

- Open your current project or create a new project.
- Select the **Project|References** menu option which will display a dialog box that lists the currently selected and available references.
- If the SocketWrench Library is listed as an available reference, select it. If it is not listed, click on the browse button and choose the CSWSKV11.DLL file.

Note that there may be another reference to the SocketWrench Control. You do not want to select this since it refers to the ActiveX control, not the library. Once the library has been referenced, you will be able to use the functions and constants detailed in the technical reference.

SocketWrench Windows Sockets Library

A general purpose TCP/IP networking library for developing client and server applications.

Reference

- [Functions](#)
- [Data Structures](#)
- [Error Codes](#)

Library Information

File Name	CSWSKV12.DLL
Version	12.0.3024.1286
LibID	99EEE601-E935-4266-B046-771A823DB47D
Import Library	CSWSKV12.LIB
Dependencies	None
Standards	RFC 768, RFC 791, RFC 793

Overview

At the core of all of the SocketTools networking libraries is the Windows Sockets API. This provides a low level interface for sending and receiving data over the Internet or a local intranet using the Transmission Control Protocol (TCP) and/or User Datagram Protocol (UDP). The SocketWrench library provides a simpler interface to the Windows Sockets API, without sacrificing features or functionality. Using SocketWrench, you can easily create client and server applications while avoiding many of the mundane tasks and common problems that developers face when building Internet applications.

This class library supports secure connections using the TLS 1.2 and TLS 1.3 protocols and can also be used to create secure, customized server applications. Both implicit and explicit TLS connections are supported, enabling the class to work with a wide variety of client and server applications without requiring third-party encryption libraries.

Requirements

The SocketTools Library Edition libraries are compatible with any programming language which supports calling functions exported from a standard dynamic link library (DLL). You should install all updates for your development tools and have the current Windows SDK installed. The minimum supported version of Visual Studio is Visual Studio 2010 with Service Pack 1 (SP1) installed. The SocketTools header files are designed to be compatible with all modern C/C++ compilers and we also provide function definitions for several additional languages such as Object Pascal and Visual Basic.

This library is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are deploying applications to Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. If you are deploying applications to Windows 8, you must have the 8.1 update installed. If you are targeting the Windows on ARM platform, the minimum required version is Windows 10 build 1709.

SocketTools includes support for both 64-bit and 32-bit Windows platforms. All current versions of Windows are 64-bit, and we include 32-bit x86 libraries to maintain compatibility with legacy development tools. SocketTools also provides native ARM64 libraries for systems running Windows on

ARM. Note that 32-bit applications on ARM use x86 emulation, and Microsoft no longer supports native 32-bit ARM (ARM32) applications. Embarcadero's C++Builder and Delphi compilers (as of version 12.3) do not support building native ARM64 applications for Windows. For developing applications targeting the Windows on ARM platform, we recommend using Visual Studio 2019 or later.

The import libraries included with SocketTools are provided in several different formats. The most commonly used format is COFF, which is available for x64, x86 and native ARM64 platforms. For compatibility with older Borland (Embarcadero) compilers, we also provide OMF libraries for x86 platforms and ELF libraries for x64 platforms. If you are using Visual Studio, you will want to use the standard COFF import libraries. Note that some languages may not require import libraries and will link to the exported functions in the DLL based on the function declaration itself.

Distribution

When you distribute your application that uses this library, it is recommended that you install the file in the same folder as your application executable. If you install the library into a shared location on the system, it is important that you distribute the correct version for the target platform and it should be registered as a shared DLL. This is a standard Windows dynamic link library, not an ActiveX component, and does not require COM registration.

SocketWrench Library Functions

Function	Description
InetAbort	Abort the connection and immediately close the socket
InetAccept	Accept a connection request from a remote host
InetAcceptEx	Accept a client connection on a listening socket with additional options
InetAsyncAccept	Accept an asynchronous connection request from a remote host
InetAsyncAcceptEx	Accept a non-blocking connection on a listening socket
InetAsyncConnect	Connect asynchronously to the specified server
InetAsyncConnectEx	Connect asynchronously to the specified server
InetAsyncListen	Listen for client connections on an asynchronous socket
InetAsyncListenEx	Listen for client connections on an asynchronous socket with additional options
InetAttachSocket	Attach the socket handle to the specified process
InetAttachThread	Attach the specified socket to another thread
InetCancel	Cancel a blocking operation
InetClientBroadcast	Write data to all other clients that are connected to the same server
InetCompareAddress	Compare two IP addresses to determine if they are identical
InetConnect	Connect to the specified server
InetConnectEx	Connect to the specified server with additional connection information
InetConnectUrl	Connect to the specified server using a URL
InetCreateSecurityCredentials	Create a new security credentials structure
InetDeleteSecurityCredentials	Delete a previously created security credentials structure
InetDetachSocket	Detach the socket handle from the current process
InetDisableEvents	Disable asynchronous event notification
InetDisableSecurity	Disable secure communication with the remote host
InetDisableTrace	Disable logging of network function calls
InetDisconnect	Disconnect from the current server
InetEnableEvents	Enable asynchronous event notification
InetEnableSecurity	Enable secure communication with the remote host
InetEnableTrace	Enable logging of network function calls to a file
InetEnumHostAliases	Return a byte array that contains all the aliases for a specified host
InetEnumNetworkAddresses	Return the list of network addresses that are configured for the local host
InetEnumServerClients	Returns a list of active client sessions established with the specified server
InetEnumServerClientsByAddress	Returns a list of active client sessions that match the specified IP address
InetEventProc	Callback method that processes events generated on the socket
InetFindClientMoniker	Returns a handle to the client socket which matches the specified moniker
InetFlush	Flush the send and receive buffers
InetFormatAddress	Convert an IP address in binary format into a printable string
InetFreezeEvents	Suspend or resume event handling by the application
InetGetAdapterAddress	Return the IP or MAC assigned to the specified network adapter

InetGetAddress	Convert an IP address string to a binary format
InetGetAddressFamily	Return the address family for the specified IP address
InetGetBlockingSocket	Return the handle for the socket which is blocked in the current thread
InetGetDefaultHostFile	Return the fully qualified path name of the host file on the local system
InetGetClientData	Returns the application defined data associated with the specified client session
InetGetClientHandle	Returns the handle for a specific client session based on its ID number
InetGetClientId	Returns the unique ID number assigned to the specified client session
InetGetClientIdleTime	Returns the amount of time the specified client session has been idle
InetGetClientMoniker	Returns the string alias associated with the specified client session
InetGetClientPriority	Returns the current priority for the specified client session
InetGetClientServer	Returns a socket handle to the server for the specified client socket
InetGetClientServerById	Returns a socket handle to the server for the specified session identifier
InetGetClientThreadId	Returns the thread ID for the specified client session
InetGetClientThreads	Returns the number of client session threads created by the server
InetGetErrorString	Return a description for the specified error code
InetGetExternalAddress	Return the external IP address assigned to the local system
InetGetHostAddress	Return the IP address assigned to the specified hostname
InetGetHostFile	Return the name of the host file
InetGetHostName	Return the hostname assigned to the specified IP address
InetGetLastError	Return the last error code
InetGetLocalAddress	Return the local IP address and port number for a socket
InetGetLocalName	Return the hostname assigned to the local system
InetGetLockedServer	Return the handle to the server which has been locked
InetGetMemoryUsage	Return the amount of memory allocated for the network connection
InetGetOption	Return the current socket options
InetGetPeerAddress	Return the IP address of the peer that the socket is connected to
InetGetPhysicalAddress	Return the media access control (MAC) address for the primary network adapter
InetGetSecurityInformation	Return information about the security characteristics of a connection
InetGetServerClient	Return the handle for the last client connection accepted by the server
InetGetServerData	Returns the application defined data associated with the specified server
InetGetServerPriority	Return the current priority assigned to the specified server
InetGetServerStackSize	Return the initial size of the stack allocated for threads created by the server
InetGetServerStatus	Returns the status of the specified server
InetGetServerThreadId	Returns the thread ID for the specified server
InetGetServiceName	Return the service name associated with a specified port number
InetGetServicePort	Return the port number associated with a service name
InetGetStatus	Report what sort of socket operation is in progress
InetGetStreamInfo	Return information about the current stream read or write operation
InetGetThreadClient	Return the handle for the client session that is being managed by the specified thread
InetGetTimeout	Return the timeout interval for blocking operations, in seconds

InetGetUrlHostName	Return the host name and port number specified in a URL
InetGetVpnAdapter	Return the name of the active Virtual Private Network adapter
InetHostNameToUnicode	Converts the canonical form of a host name to its Unicode version
InetInitialize	Initialize the library and validate the specified user license key at runtime
InetIsAddressNull	Determine if the specified IP address is a null address
InetIsAddressRoutable	Determine if the specified IP address is routable over the Internet
InetIsBlocking	Determine if the socket is performing a blocking operation
InetIsClosed	Determine if the remote host has closed its socket
InetIsConnected	Determine if the socket is connected to a remote host
InetIsListening	Determine if the socket is listening for a connection
InetIsProtocolAvailable	Determine if the specified protocol and address family are supported
InetIsReadable	Determine if data can be read from the remote process
InetIsUrgent	Determine if there is any out-of-band (OOB) data available to be read
InetIsWritable	Determine if data can be written to the remote process
InetListen	Listen for client connections on the specified socket
InetListenEx	Listen for client connections on the specified socket with additional options
InetMatchHostName	Match a host name against of list of addresses including wildcards
InetNormalizeHostName	Return the canonical form of a host name
InetPeek	Read data from the socket without removing it from the socket buffer
InetRead	Read data from the socket
InetReadEx	Read data from the socket, with extended functionality
InetReadLine	Read a line of data from the socket, storing it in a string buffer
InetReadStream	Read a stream of data from the socket
InetRegisterEvent	Register an event callback function
InetReject	Reject a pending client connection request
InetServerAsyncNotify	Enable or disable asynchronous notification of changes in server status
InetServerBroadcast	Write data to all active clients currently connected to the specified server
InetServerLock	Lock the specified server, causing all other client threads to block until it is unlocked
InetServerRestart	Restart the server, terminating all active client sessions
InetServerResume	Resume accepting client connections on the specified server
InetServerStart	Begin listening for client connections on the specified address and port
InetServerStop	Stop listening for connections and terminate all client sessions
InetServerStopEx	Stop listening for connections and wait for the server to terminate
InetServerSuspend	Suspend accepting client connections on the specified server
InetServerSuspendEx	Suspend accepting client connections and optionally reject or disconnect clients
InetServerThrottle	Limit the number of active client connections, connections per address and connection rate
InetServerUnlock	Unlock the specified server, allowing other client threads to resume execution
InetSetClientData	Associate application defined data with the specified client session
InetSetClientMoniker	Associate a unique string alias with the specified client session

InetSetClientPriority	Set the priority for the specified client session
InetSetServerData	Associate application defined data with the specified server
InetSetServerPriority	Change the priority assigned to the specified server
InetSetServerStackSize	Change the initial size of the stack allocated for threads created by the server
InetSetHostFile	Specify the name of an alternate host table
InetSetLastError	Set the last error code
InetSetOption	Set one or more options for the current socket
InetSetTimeout	Set the interval used when waiting for a blocking operation to complete
InetShutdown	Disable reception or transmission of data
InetStoreStream	Read a stream of data from the remote host and store it in a file
InetUninitialize	Terminate use of the library by the application
InetValidateCertificate	Validate the specified security certificate is installed on the local system
InetValidateHostName	Validate the specified host name and return the resolved IP address
InetWrite	Write data to the socket
InetWriteEx	Write data to the socket, with extended functionality
InetWriteLine	Write a line of data to the socket, terminated with a carriage-return and linefeed
InetWriteStream	Write a stream of data to the socket

InetAbort Function

```
INT WINAPI InetAbort(  
    SOCKET hSocket  
);
```

Immediately close the socket without waiting for any remaining data to be written out.

Parameters

hSocket

Handle to the socket.

Return Value

If the function succeeds, the return value is 0. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetAbort** function should only be used when the connection must be closed immediately before the application terminates. This function should only be used to abort client connections and should not be used with passive (listening) sockets. Server applications that need to abort an incoming client connection should use the **InetReject** function.

In most cases, the application should call the **InetDisconnect** function to gracefully close the connection to the remote host. Aborting the connection will discard any buffered data and may cause errors or result in unpredictable behavior.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetCancel](#), [InetReject](#), [InetDisconnect](#)

InetAccept Function

```
SOCKET WINAPI InetAccept(  
    SOCKET hSocket,  
    UINT nTimeout  
);
```

The **InetAccept** function is used to accept a pending client connection.

This function has been deprecated and is included for backwards compatibility. Use the **InetServerStart** function to create a server application.

Parameters

hSocket

Handle to the listening socket.

nTimeout

The number of seconds that the server will wait for the connection to complete before failing the operation.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When a connection is accepted by the server, the original listening socket continues to listen for more connections. The socket handle returned by **InetAccept** should be used to exchange information with the client.

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and accept the connection by calling **InetAccept** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

To accept a secure connection, use the **InetAcceptEx** function and specify the INET_OPTION_SECURE option.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

See Also

[InetAcceptEx](#), [InetConnect](#), [InetListen](#), [InetRegisterEvent](#), [InetReject](#), [InetServerStart](#)

InetAcceptEx Function

```
SOCKET WINAPI InetAcceptEx(  
    SOCKET hSocket,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS LpCredentials  
);
```

The **InetAcceptEx** function is used to accept a pending client connection.

This function has been deprecated and is included for backwards compatibility. Use the **InetServerStart** function to create a server application.

Parameters

- hSocket**
Handle to the socket.
- nTimeout**
The number of seconds that the server will wait for a client connection before failing the current operation.
- dwOptions**
An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.
INET_OPTION_INLINE	This option controls how urgent (out-of-band) data is handled when reading data from the socket. If set, urgent data is placed in the data stream along with non-urgent data.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has stopped the server.
INET_OPTION_SECURE	This option determines if a secure connection is negotiated with the remote host.
INET_OPTION_SECURE_FALLBACK	This option specifies the server should permit the use of less secure cipher suites for compatibility with legacy clients. If this option is specified, the server will

	allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpCredentials

Pointer to credentials structure [SECURITYCREDENTIALS](#). This may be NULL, unless *dwOptions* includes INET_OPTION_SECURE. When INET_OPTION_SECURE is used, the fields *dwSize*, *lpSzCertStore*, and *lpSzCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When a connection is accepted by the server, the original listening socket continues to listen for more connections. The socket handle returned by **InetAcceptEx** should be used to exchange information with the client.

This function will block the current thread until a connection has been established or the timeout period has elapsed. To prevent it from blocking the main user interface thread, the application should create a background worker thread and accept the connection by calling **InetAcceptEx** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetAccept](#), [InetConnect](#), [InetConnectEx](#), [InetListen](#), [InetListenEx](#), [InetRegisterEvent](#), [InetServerStart](#)

InetAsyncAccept Function

```
SOCKET WINAPI InetAsyncAccept(  
    SOCKET hSocket,  
    UINT nTimeout,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **InetAsyncAccept** function is used to accept a pending client connection.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Use the **InetServerStart** function to create a server application.

Parameters

hSocket

Handle to the listening socket.

nTimeout

The number of seconds that the server will wait for the connection to complete before failing the operation. This value is used only if *hWnd* is NULL.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When a connection is accepted by the server, the original listening socket continues to listen for more connections. The socket handle returned by **InetAsyncAccept** should be used to exchange information with the client.

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the socket handle. One or more of the following event identifiers may be sent:

Value	Description
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the process has read at least some of

	the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetAsyncConnect](#), [InetAsyncListen](#), [InetServerStart](#)

InetAsyncAcceptEx Function

```
SOCKET WINAPI InetAsyncAcceptEx(  
    SOCKET hSocket,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS lpCredentials  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **InetAsyncAcceptEx** function is used to accept a pending client connection.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Use the **InetServerStart** function to create a server application.

Parameters

hSocket

Handle to the listening socket.

nTimeout

The number of seconds that the server will wait for a client connection before failing the operation. This value is only used with blocking connections.

dwOptions

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
INET_OPTION_KEEPAIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.
INET_OPTION_INLINE	This option controls how urgent (out-of-band) data is handled when reading data from the socket. If set, urgent data is placed in the data stream along with non-urgent data.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has stopped

	the server.
INET_OPTION_SECURE	This option determines if a secure connection is negotiated with the remote host.
INET_OPTION_SECURE_FALLBACK	This option specifies the server should permit the use of less secure cipher suites for compatibility with legacy clients. If this option is specified, the server will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpCredentials

Pointer to credentials structure [SECURITYCREDENTIALS](#). This may be NULL, unless dwOptions includes INET_OPTION_SECURE. When INET_OPTION_SECURE is used, the fields **dwSize**, **lpszCertStore**, and **lpszCertName** must be defined, while other fields may be left undefined. Set **dwSize** to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When a connection is accepted by the server, the original listening socket continues to listen for more connections. The socket handle returned by **InetAsyncAccept** or **InetAsyncAcceptEx** should be used to exchange information with the client.

When a message is posted to the notification window, the low word of the **lParam** parameter contains the event identifier. The high word of **lParam** contains the low word of the error code, if an error has occurred. The **wParam** parameter contains the socket handle. One or more of the following event identifiers may be sent:

Value	Description
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.

INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.

It is recommended that you only establish an asynchronous connection if you understand the implications of doing so. In most cases, it is preferable to create a synchronous connection and create threads for each additional session if more than one active client session is required.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetAsyncAccept](#), [InetAsyncListenEx](#), [InetServerStart](#)

InetAsyncConnect Function

```
SOCKET WINAPI InetAsyncConnect(
    LPCTSTR lpszHostName,
    UINT nPort,
    UINT nProtocol,
    UINT nTimeout,
    DWORD dwOptions,
    LPSECURITYCREDENTIALS lpCredentials,
    HWND hEventWnd,
    UINT uEventMsg
);
```

The **InetAsyncConnect** function is used to establish a connection with a server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **InetConnect** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

Parameters

lpszHostName

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nPort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nProtocol

The protocol to be used when establishing the connection. This may be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout

The number of seconds to wait for a response before failing the current operation.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is constructed by using the bitwise Or operator with any of the following values:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be

	enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_DONTROUTE	This option specifies default routing should not be used. This option should not be specified unless absolutely necessary.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_RESERVEDPORT	This option specifies the socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the TLS protocol.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the <i>lpCredentials</i> parameter. If the <i>lpCredentials</i> parameter is NULL, the connection will default to using TLS 1.2 and the strongest cipher suites available. Older versions of Windows prior to Windows 7 and Windows Server 2008 R2 only support TLS 1.0 and secure connections will automatically downgrade on those platforms.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.

INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure. This parameter is only used if INET_OPTION_SECURE is specified for a TCP connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpzCertStore*, and *lpzCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When this function is called with UDP as the specified protocol, it does not actually establish a connection. Instead, it simply establishes a default destination IP address and port that is used with subsequent **InetRead** and **InetWrite** calls.

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the socket handle. One or more of the following event identifiers may be sent:

Value	Description
INET_EVENT_CONNECT	The connection to the remote host has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.

INET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.

To cancel asynchronous notification and return the socket to a blocking mode, use the **InetDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetAsyncConnectEx](#), [InetConnect](#), [InetDisableEvents](#), [InetDisconnect](#), [InetEnableEvents](#), [InetInitialize](#)

InetAsyncConnectEx Function

```
SOCKET WINAPI InetAsyncConnectEx(
    LPCTSTR lpszHostName,
    UINT nPort,
    UINT nProtocol,
    UINT nTimeout,
    DWORD dwOptions,
    LPCTSTR lpszLocalAddress,
    UINT nLocalPort,
    LPSECURITYCREDENTIALS lpCredentials,
    HWND hEventWnd,
    UINT uEventMsg
);
```

The **InetAsyncConnectEx** function is used to establish a connection with a server.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

The application should create a background worker thread and establish a connection by calling **InetConnectEx** within that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each connection.

Parameters

lpszHostName

The name of the server to connect to; this may be a fully-qualified domain name or an IP address.

nPort

The port number the server is listening on; a value of zero specifies that the default port number should be used.

nProtocol

The protocol to be used when establishing the connection. This may be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout

The number of seconds to wait for a response before failing the current operation.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are recognized:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_DONTROUTE	This option specifies default routing should not be used. This option should not be specified unless absolutely necessary.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_RESERVEDPORT	This option specifies the socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the TLS protocol.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the <i>lpCredentials</i> parameter. If the <i>lpCredentials</i> parameter is NULL, the connection will default to using TLS 1.2 and the strongest cipher suites available. Older versions of Windows prior to Windows 7 and Windows Server 2008 R2 only support TLS 1.0 and secure connections will automatically downgrade on those platforms.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will

	allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpzLocalAddress

A null-terminated string that specifies the local IP address that the socket should be bound to. If this parameter is NULL, then an appropriate address will automatically be used. A specific address should only be used if it is required by the application.

nLocalPort

The local port number that the socket should be bound to. If this parameter is set to zero, then an appropriate port number will automatically be used. A specific port number should only be used if it is required by the application.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure. This parameter is only used if INET_OPTION_SECURE is specified for a TCP connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields ***dwSize***, ***lpzCertStore***, and ***lpzCertName*** must be defined, while other fields may be left undefined. Set ***dwSize*** to the size of the **SECURITYCREDENTIALS** structure.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the ***lParam*** parameter contains the event identifier. The high word of ***lParam*** contains the low word of the error code, if an error has occurred. The ***wParam*** parameter contains the socket handle. One or more of the following event identifiers may be sent:

Value	Description
INET_EVENT_CONNECT	The connection to the remote host has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will be posted if an error has occurred.
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to read by the calling process. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetAsyncConnect](#), [InetConnect](#), [InetDisableEvents](#), [InetDisconnect](#), [InetEnableEvents](#), [InetInitialize](#)

InetAsyncListen Function

```
SOCKET WINAPI InetAsyncListen(  
    LPCTSTR lpszLocalAddress,  
    UINT nLocalPort,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **InetAsyncListen** function creates an listening socket.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Use the **InetServerStart** function to create a server application.

Parameters

lpszLocalAddress

A null-terminated string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The socket option INET_OPTION_REUSEADDRESS is enabled by default when calling the **InetAsyncListen** function. This allows an application to re-use a local address and port number when

creating the listening socket. If this behavior is not desired, use the **InetAsyncListenEx** function instead.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

When a message is posted to the notification window, the low word of the **lParam** parameter contains the event identifier. The high word of **lParam** contains the low word of the error code, if an error has occurred. The **wParam** parameter contains the socket handle. One or more of the following event identifiers may be sent:

Value	Description
INET_EVENT_ACCEPT	The process has received a connection request from a client and should accept the connection using the InetAsyncAccept function. This event is only generated for server applications which have created an asynchronous socket using the InetAsyncListen function.

To cancel asynchronous notification and return the socket to a blocking mode, use the **InetDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetAsyncAccept](#), [InetAsyncListenEx](#), [InetDisableEvents](#), [InetEnableEvents](#), [InetInitialize](#), [InetListen](#), [InetServerStart](#)

InetAsyncListenEx Function

```
SOCKET WINAPI InetAsyncListenEx(  
    LPCTSTR lpszLocalAddress,  
    UINT nLocalPort,  
    UINT nBacklog,  
    DWORD dwOptions,  
    HWND hEventWnd,  
    UINT uEventMsg  
);
```

The **InetAsyncListenEx** function creates an listening socket.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Use the **InetServerStart** function to create a server application.

Parameters

lpszLocalAddress

A null-terminated string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

nBacklog

The maximum length of the queue allocated for pending client connections. A value of zero specifies that the size of the queue should be set to a maximum reasonable value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are supported:

Value	Description
INET_OPTION_NONE	No option specified. If the address and port number are in use by another application or a closed socket which was listening on this port is still in the TIME_WAIT state, the function will fail.
INET_OPTION_REUSEADDRESS	This option enables a server application to listen for connections using the specified address and port number even if they were in use recently. This is typically used to enable an application to close the listening socket and

	immediately reopen it without getting an error that the address is in use.
INET_OPTION_EXCLUSIVE	This option specifies the local address and port number is for the exclusive use by the current process, preventing another application from forcibly binding to the same address. If another process has already bound a socket to the address provided by the caller, this function will fail.
INET_OPTION_RESERVEDPORT	This option specifies the listening socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges and firewall rules that will permit a client to establish a connection with the service.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has stopped the server.

hEventWnd

The handle to the event notification window. This window receives messages which notify the application of various asynchronous socket events that occur.

uEventMsg

The message identifier that is used when an asynchronous socket event occurs. This value should be greater than WM_USER as defined in the Windows header files.

Return Value

If the function succeeds, the return value is a handle to the socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

If the INET_OPTION_REUSEADDRESS option is not specified, an error may be returned if a listening socket was recently created for the same local address and port number. By default, once a listening socket is closed there is a period of time that all applications must wait before the address can be reused (this is called the TIME_WAIT state). The actual amount of time depends on the operating system and configuration parameters, but is typically two to four minutes. Specifying this option enables an application to immediately re-use a local address and port number that was previously in use.

If the INET_OPTION_EXCLUSIVE option is specified, the local address and port number cannot be used by another process until the listening socket is closed. This can prevent another application from forcibly binding to the same listening address as your server. This option can be useful in determining whether or not another process is already bound to the address you wish to use, but it may also

prevent your server application from restarting immediately, regardless if the INET_OPTION_REUSEADDRESS option has also been specified.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the socket handle. One or more of the following event identifiers may be sent:

Value	Description
INET_EVENT_ACCEPT	An incoming client connection is pending. The connection will be assigned to a new socket. This event is only generated if the socket is in asynchronous mode.

To cancel asynchronous notification and return the socket to a blocking mode, use the **InetDisableEvents** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetAsyncAccept](#), [InetAsyncAcceptEx](#), [InetAsyncListen](#), [InetDisableEvents](#), [InetEnableEvents](#), [InetListenEx](#), [InetServerStart](#)

InetAttachSocket Function

```
SOCKET WINAPI InetAttachSocket(  
    SOCKET hSocket  
    DWORD dwProcessId  
);
```

The **InetAttachThread** function attaches the specified socket handle to another thread.

Parameters

hSocket

Handle to the socket.

dwProcessId

The process ID for the process that currently owns the socket handle. This value may be zero to specify the current process.

Return Value

If the function succeeds, the return value is the handle to the attached socket. If the function fails, the return value is `INVALID_SOCKET`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetAttachSocket** function enables an application to attach an external socket handle to the current thread and initializes it for use by the library. An external socket is any socket handle created directly by the Windows Sockets API or by a third-party library. If the *dwProcessId* parameter is zero or specifies the current process ID, then the function checks to see if the socket was created by this library. If it was, then its owner context is switched to the current thread; if the socket was created externally, then it is initialized for use by the library and attached to the current thread.

If the *dwProcessId* parameter specifies another process, the socket will be duplicated into the current process, attached to the current thread and the original socket handle will be closed in the other process. This enables an application to effectively take control of a connection created by another process. The original socket handle must be inheritable by the by the current process and must be an actual Windows socket handle, not a pseudo-handle. This functionality is only supported on Windows NT 4.0 and later versions of the operating system with the Microsoft TCP/IP stack. Note that Layered Service Providers (LSPs) may interfere with the ability to inherit handles across processes.

The attached socket is initialized in a blocking state, even if was originally using asynchronous socket events. If the application requires that the socket use events, it must explicitly call **InetEnableEvents** using the handle returned by this function.

In most cases, the handle returned by this function will be the same value as the *hSocket* parameter, however an application should never make the assumption that this will be the case. If

InetAttachSocket returns a socket handle that has a different value than the *hSocket* parameter, this indicates that the original handle has been destroyed and should never be used in subsequent function calls.

This function should never be used with a secure socket connection because the attached socket will not have the security context required to encrypt and decrypt the data exchanged with the remote host.

Example

To demonstrate how to pass sockets between processes, this example will use two programs; one

acting as a server to listen for client connections and accept them, the other inheriting the client socket and echoing back anything the client sends. The first program will create the listening socket, and when a client connects, it will call **CreateProcess** to create a new child process to handle that connection.

```
SOCKET hServer;
SOCKET hClient;

// Initialize the library

if (!InetInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    return;

// Listen for incoming client connections

if ((hServer = InetListen(NULL, nLocalPort)) == INVALID_SOCKET)
    return;

while (TRUE)
{
    hClient = InetAccept(hServer, 10);

    if (hClient == INVALID_SOCKET)
    {
        // If InetAccept has timed-out, then simply loop back and attempt
        // to continue accepting connections; otherwise, exit the loop

        if (InetGetLastError() != ST_ERROR_OPERATION_TIMEOUT)
            break;
    }
    else
    {
        STARTUPINFO si;
        PROCESS_INFORMATION pi;
        CHAR szCommandLine[512];
        BOOL bResult;

        // Detach the socket, which will free the memory that the library
        // has allocated for it without actually destroying the socket handle;
        // the child process will close the handle in this process when it
        // attaches to it
        InetDetachSocket(hClient, 0);

        // Initialize the STARTUPINFO structure
        ZeroMemory(&si, sizeof(si));

        // Create the command line arguments, passing the current
        // process ID and socket handle to the new process

        wsprintf(szCommandLine, "%s %lu %lu",
                lpzAppName,
                (DWORD)GetCurrentProcessId(),
                (DWORD)hClient);

        // Create the child process

        bResult = CreateProcess(NULL,
                                szCommandLine,
                                NULL, NULL,
```

```

        TRUE,
        CREATE_DEFAULT_ERROR_MODE,
        NULL, NULL,
        &si, &pi);

    if (!bResult)
        InetDisconnect(hClient);
}

InetDisconnect(hServer);
InetUninitialize();

```

The second program attaches to the socket handle that was passed to it by the parent process. It goes into a loop, reading any data sent to it by the client and sending the same data back. When the client disconnects, the **InetRead** function will return 0, it will exit the loop and the process will terminate.

```

SOCKET hSocket;
SOCKET hClient;
DWORD dwProcessId;

// Initialize the library

if (!InetInitialize(CSTOOLS12_LICENSE_KEY, NULL))
    return;

// Process command line arguments that were passed to us
// by the server process

dwProcessId = (DWORD)atol(argv[1]);
hClient = (SOCKET)atol(argv[2]);

// Attach to the hClient socket that the server passed
// to us; this will close the socket in the server process

hSocket = InetAttachSocket(hClient, dwProcessId);

if (hSocket != INVALID_SOCKET)
{
    BYTE cBuffer[512];
    int nRead;

    do
    {
        // Read any data sent to us by the client
        nRead = InetRead(hSocket, cBuffer, sizeof(cBuffer));

        // Echo the data we have read back to the client
        if (nRead > 0)
            InetWrite(hSocket, cBuffer, nRead);
    }
    while (nRead > 0);

    InetDisconnect(hSocket);
}

InetUninitialize();

```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetAttachThread](#), [InetDetachSocket](#), [InetInitialize](#)

InetAttachThread Function

```
DWORD WINAPI InetAttachThread(  
    SOCKET hSocket  
    DWORD dwThreadId  
);
```

The **InetAttachThread** function attaches the specified socket handle to another thread.

Parameters

hSocket

Handle to the socket.

dwThreadId

The ID of the thread that will become the new owner of the handle. A value of zero specifies that the current thread should become the owner of the socket handle.

Return Value

If the function succeeds, the return value is the thread ID of the previous owner. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

When a socket handle is created, it is associated with the current thread that created it. Normally, if another thread attempts to perform an operation using that handle, an error is returned since it does not own the handle. This is used to ensure that other threads cannot interfere with an operation being performed by the owner thread. In some cases, it may be desirable for one thread in an application to create the socket, and then pass that handle to another worker thread. The **InetAttachThread** function can be used to change the ownership of the handle to the new worker thread. By preserving the return value from the function, the original owner of the handle can be restored before the worker thread terminates.

This function should be called by the new thread immediately after it has been created, and if the new thread does not release the handle itself, the ownership of the handle should be restored to the parent thread before it terminates. Under no circumstances should **InetAttachThread** be used to forcibly release a handle allocated by another thread while a blocking operation is in progress. To cancel an operation, use the **InetCancel** function and then release the handle after the blocking function exits and control is returned to the current thread.

Note that the *dwThreadId* parameter is presumed to be a valid thread ID and no checks are performed to ensure that the thread actually exists. Specifying an invalid thread ID will orphan the handle until the **InetUninitialize** function is called.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetAccept](#), [InetAcceptEx](#), [InetConnect](#), [InetConnectEx](#), [InetDisconnect](#)

InetCancel Function

```
INT WINAPI InetCancel(  
    SOCKET hSocket  
);
```

The **InetCancel** function cancels a blocking operation.

Parameters

hSocket

Handle to the socket.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

When the **InetCancel** function is called, the blocking function will not immediately fail. An internal flag is set which causes the blocking operation to exit with an error. This means that the application cannot cancel an operation and immediately perform some other operation. Instead it must allow the calling stack to unwind, returning back to the blocking operation before making any further function calls.

This function is typically called from within an event handler to signal that the current blocking operation should stop. It may also be used to cancel a blocking operation that is occurring on another thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetAbort](#)

InetClientBroadcast Function

```
INT WINAPI InetClientBroadcast(  
    SOCKET hClient,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **InetClientBroadcast** function sends data to all other clients that are connected to the same server.

Parameters

hClient

The socket handle.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server clients.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of clients that the data was sent to. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetClientBroadcast** function sends the contents of the buffer to all other clients that are connected to the same server as the specified client. This function will block until all clients have been sent a copy of the data. There is no guarantee in which order the clients will receive and process the data that has been broadcast.

This function can only be used with client sessions created as part of the server interface and cannot be used with standard sockets created using the **InetConnect** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetServerBroadcast](#), [InetWrite](#), [InetWriteLine](#)

InetCompareAddress Function

```
BOOL WINAPI InetCompareAddress(  
    LPINTERNET_ADDRESS LpAddress1,  
    LPINTERNET_ADDRESS LpAddress2  
);
```

The **InetCompareAddress** function compares two Internet addresses in a binary format.

Parameters

LpAddress1

A pointer to an INTERNET_ADDRESS structure that contains the first IP address to be compared.

LpAddress2

A pointer to an INTERNET_ADDRESS structure that contains the second IP address to be compared.

Return Value

If the function succeeds and the two addresses are identical, the return value is non-zero. If the function fails or the two addresses are not identical, the return value is zero. If either parameter is NULL, or the address family for the two addresses are not the same, the last error code will be updated. If the addresses are valid and in the same address family, but are not identical, the last error code will be set to NO_ERROR.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetHostAddress](#), [InetGetLocalAddress](#), [InetGetPeerAddress](#), [INTERNET_ADDRESS](#)

InetConnect Function

```
SOCKET WINAPI InetConnect(  
    LPCTSTR lpszHostName,  
    UINT nRemotePort,  
    UINT nProtocol,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS lpCredentials  
);
```

The **InetConnect** function is used to establish a connection with a server.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address of the system you want to connect with. This parameter cannot be a URL and must only specify the name of the remote host. If this parameter is NULL or an empty string, the function will fail with an error indicating the host name is invalid.

nRemotePort

The port number used to establish the connection. Valid port numbers range in value from 1 through 65535 and a value outside of this range will cause the function to fail. Port numbers in the range of 49152 and 65535 are referred to as dynamic ports and are generally reserved for private use by client applications. You cannot specify a port number of zero when establishing an outbound connection.

nProtocol

The protocol to be used when establishing the connection. This may be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout

The number of seconds to wait for the connection to complete before failing the current operation.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is constructed by using the bitwise Or operator with any of the following values:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_DONTROUTE	This option specifies default routing should not be

	used. This option should not be specified unless absolutely necessary.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_RESERVEDPORT	This option specifies the socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the TLS protocol.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the <i>lpCredentials</i> parameter. If the <i>lpCredentials</i> parameter is NULL, the connection will default to using TLS 1.2 or later and the strongest cipher suites available.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6

	address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure. This parameter is only used if INET_OPTION_SECURE is specified for a TCP connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpzCertStore*, and *lpzCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to a socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The *lpzHostName* parameter must specify a valid host name or IP address. Host names are resolved into an IP address by first checking the local hosts file and if the name is not found, a name server query will be performed to determine the IP address. If the Unicode version of this function is called and the host name includes non-ASCII characters, the host name will be automatically converted to an ASCII compatible format. Refer to the **InetNormalizeHostName** function for more information. To establish a connection using a URL rather than a host name, use the **InetConnectUrl** function.

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **InetConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If you use the INET_OPTION_SECURE option to enable a secure connection, the connection will always use implicit TLS. This means a secure session will be initiated immediately after the socket connection has been established with the server. A common example of a service which uses implicit TLS is the HTTPS protocol. Another type of secure connection is one that uses explicit TLS. This is when the client establishes a normal (non-secure) connection with the server and then explicitly switches to using a secure connection, typically by sending a command. If the server you are connecting to requires explicit TLS, you should not specify the INET_OPTION_SECURE option. Instead, connect without this option and then call the **InetEnableSecurity** function when you are ready to initiate the TLS handshake.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the INET_OPTION_FREETHREAD option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected

behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

When this function is called with UDP as the specified protocol, it does not actually establish a connection in the same way that a TCP stream connection is created. Instead, it simply establishes a default destination IP address and port that is used with subsequent **InetRead** and **InetWrite** calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetConnectEx](#), [InetConnectUrl](#), [InetDisconnect](#), [InetEnableSecurity](#), [InetInitialize](#), [InetRead](#), [InetRegisterEvent](#), [InetWrite](#)

InetConnectEx Function

```
SOCKET WINAPI InetConnectEx(  
    LPCTSTR lpszHostName,  
    UINT nRemotePort,  
    UINT nProtocol,  
    UINT nTimeout,  
    DWORD dwOptions,  
    LPCTSTR lpszLocalAddress,  
    UINT nLocalPort,  
    LPSECURITYCREDENTIALS lpCredentials  
);
```

The **InetConnectEx** function is used to establish a connection with a server.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address of the system you want to connect with. This parameter cannot be a URL and must only specify the name of the remote host. If this parameter is NULL or an empty string, the function will fail with an error indicating the host name is invalid.

nRemotePort

The port number used to establish the connection. Valid port numbers range in value from 1 through 65535 and a value outside of this range will cause the function to fail. Port numbers in the range of 49152 and 65535 are referred to as dynamic ports and are generally reserved for private use by client applications. You cannot specify a port number of zero when establishing an outbound connection.

nProtocol

The protocol to be used when establishing the connection. This may be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This is the default protocol.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. Note that UDP is unreliable in that there is no way for the sender to know that the receiver has actually received the datagram.

nTimeout

The number of seconds to wait for the connection to complete before failing the operation.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is constructed by using the bitwise Or operator with any of the following values:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for

	stream sockets.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_RESERVEDPORT	This option specifies the socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects connections using either the TLS protocol.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The specific version of TLS and other security related options are provided in the <i>lpCredentials</i> parameter. If the <i>lpCredentials</i> parameter is NULL, the connection will default to using TLS 1.2 or later and the strongest cipher suites available.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a

	connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

lpzLocalAddress

A null-terminated string that specifies the local IP address that the socket should be bound to. If this parameter is NULL, then an appropriate address will automatically be used. A specific address should only be used if it is required by the application.

nLocalPort

The local port number that the socket should be bound to. If this parameter is set to zero, then an appropriate port number will automatically be used. A specific port number should only be used if it is required by the application.

lpCredentials

A pointer to a [SECURITYCREDENTIALS](#) structure. This parameter is only used if INET_OPTION_SECURE is specified for a TCP connection. This parameter may be NULL, in which case no client credentials will be provided to the server. If client credentials are required, the fields *dwSize*, *lpzCertStore*, and *lpzCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a handle to a socket. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

The *lpzHostName* parameter must specify a valid host name or IP address. Host names are resolved into an IP address by first checking the local hosts file and if the name is not found, a name server query will be performed to determine the IP address. If the Unicode version of this function is called and the host name includes non-ASCII characters, the host name will be automatically converted to an ASCII compatible format. Refer to the **InetNormalizeHostName** function for more information. To establish a connection using a URL rather than a host name, use the **InetConnectUrl** function.

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **InetConnect** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

If you use the INET_OPTION_SECURE option to enable a secure connection, the connection will always use implicit TLS. This means a secure session will be initiated immediately after the socket connection has been established with the server. A common example of a service which uses implicit TLS is the HTTPS protocol. Another type of secure connection is one that uses explicit TLS. This is when the client establishes a normal (non-secure) connection with the server and then explicitly switches to using a secure connection, typically by sending a command. If the server you are connecting to requires explicit TLS, you should not specify the INET_OPTION_SECURE option. Instead, connect without this option and then call the **InetEnableSecurity** function when you are ready to initiate the TLS handshake.

The *dwOptions* argument can be used to specify the threading model that is used by the library when

a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the `INET_OPTION_FREETHREAD` option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

When this function is called with UDP as the specified protocol, it does not actually establish a connection in the same way that a TCP stream connection is created. Instead, it simply establishes a default destination IP address and port that is used with subsequent **InetRead** and **InetWrite** calls.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetConnect](#), [InetConnectUrl](#), [InetDisconnect](#), [InetEnableSecurity](#), [InetInitialize](#), [InetRead](#), [InetRegisterEvent](#), [InetWrite](#)

InetConnectUrl Function

```
SOCKET WINAPI InetConnectUrl(  
    LPCTSTR lpszUrl,  
    UINT nTimeout,  
    DWORD dwOptions  
);
```

The **InetConnectUrl** function is used to establish a TCP connection with a server using the information provided in a URL.

Parameters

lpszUrl

A null-terminated string which specifies a URL used when establishing the connection. This parameter cannot be NULL or point to an empty string. If a non-standard URI scheme is used, the port number must be explicitly specified or the function will fail. See the remarks below for more information on the format supported by this function.

nTimeout

The number of seconds to wait for the connection to complete before failing the current operation. If this value is zero, a default timeout period will be used.

dwOptions

An unsigned integer used to specify one or more socket options. This parameter is constructed by using the bitwise Or operator with any of the following values:

Value	Description
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This option is not necessary for most connections, particularly when the client will not be connected to the server for an extended period of time.
INET_OPTION_NODELAY	This option disables the Nagle algorithm. By default, small amounts of data written to the socket are buffered, increasing efficiency and reducing network congestion. However, this buffering can negatively impact the responsiveness of certain applications. This option disables this buffering and immediately sends data packets as they are written to the socket.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_TRUSTEDSITE	This option specifies the server is trusted. The server certificate will not be validated and the connection will always be permitted. This option only affects secure

	connections using the TLS protocol.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the remote host. The connection will always default to using TLS 1.2 or later and the strongest cipher suites available on the client platform. This option may be automatically enabled if the URL scheme specifies a service which requires a secure connection. See the remarks below for more information.
INET_OPTION_SECURE_FALLBACK	This option specifies the client should permit the use of less secure cipher suites for compatibility with legacy servers. If this option is specified, the client will allow connections using TLS 1.0 and cipher suites that use RC4, MD5 and SHA1.
INET_OPTION_PREFER_IPV6	This option specifies the client should prefer the use of IPv6 if the server hostname can be resolved to both an IPv6 and IPv4 address. This option is ignored if the local system does not have IPv6 enabled, or when the hostname can only be resolved to an IPv4 address. If the server hostname can only be resolved to an IPv6 address, the client will attempt to establish a connection using IPv6 regardless if this option has been specified.
INET_OPTION_FREETHREAD	This option specifies the socket returned by this function may be used by any thread, and is not limited to the thread which created it. The application is responsible for ensuring that access to the socket is synchronized across multiple threads.

Return Value

If the function succeeds, the return value is a handle to a socket. If the function fails, the return value is `INVALID_SOCKET`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetConnectUrl** function provides a simplified interface which can be used to establish a connection using a URL. This function can only be used to establish connections using TCP and does not currently support the use of URLs to connect with a service which uses UDP. The general format of the URL should look like this:

```
[scheme]:// [[username : password] @] hostname [:port] / [path;parameters ...]
```

This function recognizes most standard URI schemes which use this format. The host name and port number specified in the URL will be used to establish a connection and the remaining information will be discarded. If the URL does not explicitly specify a port number, the default port number associated with the scheme will be used as the default value. For example, consider the following:

```
https://www.example.com
```

In this example, there is no port number specified; instead, the default port for the **https://** scheme would be used, which is port 443. The host name **www.example.com** would be resolved into an IP

address and the connection established on port 443. This function will also recognize a simpler format which only includes the host name and port number without a URI scheme, such as:

www.example.com:443

When used in this way, the port number must always be provided. Without a URI scheme or an explicit port number, the function cannot determine what port number should be used when establishing the connection. The same also applies if a custom, non-standard URI scheme is provided which is not recognized.

If the URI scheme specifies a secure protocol which requires implicit TLS, this function will automatically enable the `INET_OPTION_SECURE` option. For example, providing a URL which uses the **https://** scheme will automatically enable a secure connection regardless if the *dwOptions* parameter includes that option. If a URI scheme is used in conjunction with a port number associated with a secure service, security will also be enabled for that connection. For example:

http://www.example.com:443

The standard **http://** scheme is used which does not indicate a secure connection. However, since port 443 is the standard port designated for a secure HTTP connection, a secure connection will be enabled by default, even if `INET_OPTION_SECURE` has not been specified by the caller. Alternatively, if a custom port number is specified in the URL or the scheme is not recognized as one which requires implicit TLS, security options will not be automatically enabled for the connection.

The host name portion of the URL can be specified as either a domain name or an IP address. Because an IPv6 address can contain colon characters, you must enclose the entire address in bracket `[]` characters. If this is not done, this function will return an error indicating the port number is invalid. For example, the URL **https://[2001:db8:0:0:1::128]/** uses an IPv6 host address and this would be considered valid. Without the brackets, this URL would not be accepted.

Important: The URL provided to this function will only be used to establish a connection with a server. This is a general purpose function which does not enable support for any particular application protocol and all implementation details are the responsibility of your application. If you require higher-level support for a specific Internet protocol, the SocketTools API provides comprehensive collection of higher-level functions which can be used to access those services.

If you use the `INET_OPTION_SECURE` option to enable a secure connection, the connection will always use implicit TLS. This means a secure session will be initiated immediately after the socket connection has been established with the server. A common example of a service which uses implicit TLS is the HTTPS protocol. Another type of secure connection is one that uses explicit TLS. This is when the client establishes a normal (non-secure) connection with the server and then explicitly switches to using a secure connection, typically by sending a command. If the server you are connecting to requires explicit TLS, you should not specify the `INET_OPTION_SECURE` option. Instead, connect without this option and then call the **InetEnableSecurity** function when you are ready to initiate the TLS handshake.

To prevent this function from blocking the main user interface thread, the application should create a background worker thread and establish a connection by calling **InetConnectUrl** in that thread. If the application requires multiple simultaneous connections, it is recommended you create a worker thread for each client session.

The *dwOptions* argument can be used to specify the threading model that is used by the library when a connection is established. By default, the handle is initially attached to the thread that created it. From that point on, until the handle is released, only the owner may call functions using that handle. The ownership of the handle may be transferred from one thread to another using the **InetAttachThread** function.

Specifying the `INET_OPTION_FREETHREAD` option enables any thread to call any function using the socket handle, regardless of which thread created it. It is important to note that this option disables certain internal safety checks which are performed by the library and may result in unexpected behavior unless access to the socket is synchronized. If one thread calls a function in the library, it must ensure that no other thread will call another function at the same time using the same socket handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetConnect](#), [InetDisconnect](#), [InetEnableSecurity](#), [InetGetUrlHostName](#), [InetInitialize](#), [InetRead](#), [InetRegisterEvent](#), [InetWrite](#)

InetCreateSecurityCredentials Function

```
BOOL WINAPI InetCreateSecurityCredentials(  
    DWORD dwProtocol,  
    DWORD dwOptions,  
    LPCTSTR lpszUserName,  
    LPCTSTR lpszPassword,  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertName,  
    LPVOID lpvReserved,  
    LPSECURITYCREDENTIALS *LppCredentials  
);
```

The **InetCreateSecurityCredentials** function allocates a new security credentials structure.

Parameters

dwProtocol

A bitmask of supported security protocols. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or more options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

lpszUserName

A null-terminated string which specifies the certificate owner's username. A value of NULL specifies that no username is required. Currently this parameter is not used and any value specified will be ignored.

lpszPassword

A null-terminated string which specifies the certificate owner's password. A value of NULL specifies that no password is required. This parameter is only used if a PKCS #12 (PFX) certificate file is specified and that certificate has been secured with a password. This value will be ignored the current user or local machine certificate store is specified.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by

	entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

lpvReserved

Pointer reserved for future use. Set it to NULL when using this function.

lppCredentials

Pointer to an [LPSECURITYCREDENTIALS](#) pointer. The memory for the credentials structure will be allocated by this function and must be released by calling the **InetDeleteSecurityCredentials** function when it is no longer needed. The pointer value must be set to NULL before the function is called. It is important to note that this is a pointer to a pointer variable, not a pointer to the SECURITYCREDENTIALS structure itself.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call the **InetGetLastError** function.

Remarks

The structure that is created by this function may be used as client credentials when establishing a secure connection. This is particularly useful for programming languages other than C/C++ which may not support C structures or pointers. The pointer to the SECURITYCREDENTIALS structure can be declared as an unsigned integer variable which is passed by reference to this function, and then passed by value to the **InetConnect** function.

A client application only needs to call this function if the server requires that the client provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user,

or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The *dwOptions* parameter should be set to CREDENTIAL_STORE_FILENAME and the *lpzCertStore* parameter should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate was created, it must be provided in the *lpzPassword* parameter or the library will be unable to access the certificate.

Note that the *lpzUserName* and *lpzPassword* parameters are values which are used to access the certificate store or client certificate. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session. In most cases, these parameters will be NULL.

Example

```
LPSECURITYCREDENTIALS lpSecCred = NULL;
InetCreateSecurityCredentials(SEcurity_PROTOCOL_DEFAULT,
                             CREDENTIAL_STORE_CURRENT_USER,
                             NULL,
                             NULL,
                             strCertStore,
                             strCertName,
                             NULL,
                             &lpSecCred);

hAcceptSocket = InetAsyncAcceptEx(hListenSocket,
                                   nTimeout,
                                   dwOptions,
                                   lpSecCred,
                                   hEventWnd,
                                   uEventMsg);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetAcceptEx](#), [InetAsyncAcceptEx](#), [InetAsyncConnectEx](#), [InetConnectEx](#), [InetDeleteSecurityCredentials](#), [InetValidateCertificate](#)

InetDeleteSecurityCredentials Function

```
VOID WINAPI InetDeleteSecurityCredentials(  
    LPSECURITYCREDENTIALS *LppCredentials  
);
```

The **InetDeleteSecurityCredentials** function deletes an existing SECURITYCREDENTIALS structure.

Parameters

lppCredentials

Pointer to an LPSECURITYCREDENTIALS pointer. On exit from the function, the pointer value will be NULL.

Return Value

None.

Example

```
if (lpSecCred != NULL)  
    InetDeleteSecurityCredentials(&lpSecCred);
```

Remarks

This function can be used to release the memory allocated to the client or server credentials after a secure connection has been established.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetCreateSecurityCredentials](#), [InetValidateCertificate](#)

InetDetachSocket Function

```
BOOL WINAPI InetDetachSocket(  
    SOCKET hSocket  
    DWORD dwThreadId  
);
```

The **InetDetachSocket** function detaches the specified socket from the current process.

Parameters

hSocket

Handle to the socket.

dwThreadId

The ID of the thread that owns the socket handle. A value of zero specifies that the current thread is the owner.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetDetachSocket** function will release the memory that the library has allocated for the socket without destroying the socket handle. After the function returns, the socket can no longer be used with other functions in the library, however the socket handle remains valid. This is typically used when passing a socket handle between processes, where the parent process detaches the socket prior to creating the child process. The child then calls **InetAttachSocket** to attach the socket handle to its own process.

This function should never be used with a secure socket connection because detaching a secure socket will force the security context for that session to be released. If the socket is attached to another process, it will not have the security context originally created when the connection was established and will be unable to encrypt or decrypt the data stream.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: csoskv12.lib

See Also

[InetAttachThread](#), [InetAttachSocket](#), [InetDisconnect](#)

InetDisableEvents Function

```
INT WINAPI InetDisableEvents(  
    SOCKET hSocket  
);
```

The **InetDisableEvents** function disables the event notification mechanism, preventing subsequent event notification messages from being posted to the application's message queue.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Parameters

hSocket

The socket handle.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

This function affects both event notification and event callbacks. Any outstanding events in the message queue should be ignored by the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetEnableEvents](#), [InetFreezeEvents](#), [InetRegisterEvent](#)

InetDisableSecurity Function

```
INT WINAPI InetDisableSecurity(  
    SOCKET hSocket,  
    DWORD dwReserved  
);
```

The **InetDisableSecurity** function disables a secure session with the remote host.

Parameters

hSocket

The socket handle.

dwReserved

Reserved parameter. This value should always be zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetDisableSecurity** function disables a secure session, with subsequent calls to **InetRead** and **InetWrite** sending and receiving unencrypted data. It is important to note that because this function sends a shutdown message to terminate the secure session, this may cause connection to be closed by the remote host.

This function does not close the socket. Use the **InetDisconnect** function to close the socket and release the resources allocated for the current session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetCreateSecurityCredentials](#), [InetDeleteSecurityCredentials](#), [InetEnableSecurity](#)

InetDisableTrace Function

```
BOOL WINAPI InetDisableTrace();
```

The **InetDisableTrace** function disables the logging of network function calls.

Parameters

None.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

This function will disable output to the log file for the current thread or process, depending on the flags used when logging was enabled. If this function is not called, the resources that were allocated for the logging session will be released when the process ends.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetEnableTrace](#)

InetDisconnect Function

```
INT WINAPI InetDisconnect(  
    SOCKET hSocket  
);
```

The **InetDisconnect** function terminates the connection, closing the socket and releasing the memory allocated for the session.

Parameters

hSocket

The socket handle.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

Once the connection has been terminated, the socket handle is no longer valid and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new connection is established. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

After a socket is closed, it will go into a TIME-WAIT state which prevents an application from using the same source and destination address and port numbers bound to that socket for a brief period of time, typically two to four minutes. This is normal behavior designed to prevent delayed or misrouted packets of data from being read by a subsequent connection. This can have an impact on an application that rapidly connects and disconnects over a short period of time because it can exhaust the pool of ephemeral ports.

If this function is called using a server socket handle returned by the **InetServerStart** function, all active client connections will be disconnected, the listening socket will be closed and the server thread will terminate. If this function is called with a client socket handle allocated by the server, it will terminate the client connection and the thread that manages it.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetConnect](#), [InetConnectEx](#), [InetServerStart](#), [InetUninitialize](#)

InetEnableEvents Function

```
INT WINAPI InetEnableEvents(  
    SOCKET hSocket,  
    HWND hEventWnd,  
    UINT nEventMsg  
);
```

The **InetEnableEvents** function enables event notifications using Windows messages.

This function has been deprecated and may be unavailable in future releases. It was designed for use in legacy single-threaded applications and requires the application to have a message pump to process event messages. It should not be used with applications which are designed to execute as a service or those which do not have a graphical user interface.

Applications should use the **InetRegisterEvent** function to register an event handler which is invoked when an event occurs.

Parameters

hSocket

The socket handle.

hEventWnd

Handle to the event notification window. This window receives a user-defined message which specifies the event that has occurred. If this value is NULL, event notification is disabled.

nEventMsg

An unsigned integer which specifies the user-defined message that is sent when an event occurs. This parameter's value must be greater than the value of WM_USER.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

When a message is posted to the notification window, the low word of the *lParam* parameter contains the event identifier. The high word of *lParam* contains the low word of the error code, if an error has occurred. The *wParam* parameter contains the socket handle. One or more of the following event identifiers may be sent:

Value	Description
INET_EVENT_ACCEPT	The process has received a connection request from a client and should accept the connection using the InetAsyncAccept function. This event is only generated for server applications which have created an asynchronous socket using the InetAsyncListen function.
INET_EVENT_CONNECT	The connection to the remote host has completed. The high word of the <i>lParam</i> parameter should be checked, since this notification message will also be posted if an error has occurred.
INET_EVENT_DISCONNECT	The remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	Data is available to read by the calling process. No additional

	messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	The process can now write data. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_TIMEOUT	The process has timed-out waiting for a blocking operation to complete. This event is only generated for synchronous sockets.
INET_EVENT_CANCEL	The application has canceled a blocking operation. This event is fired once an operation has been terminated by the InetCancel function, and control has been returned to the calling process.

This function cannot be used with sockets that are created by the SocketWrench server interface. Those sockets are managed separately in their own thread, and event notifications are handled inside the callback function specified when the server is created using the **InetServerStart** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetDisableEvents](#), [InetFreezeEvents](#), [InetRegisterEvent](#)

InetEnableSecurity Function

```
INT WINAPI InetEnableSecurity(  
    SOCKET hSocket,  
    DWORD dwOptions,  
    LPSECURITYCREDENTIALS lpCredentials  
);
```

The **InetEnableSecurity** function enables a secure session with the remote host.

Parameters

hSocket

The socket handle.

dwOptions

An unsigned integer value that specifies additional security options. It may have a value of zero or one of the following options:

Value	Description
INET_SECURE_CLIENT	The certificate specified by the <i>lpCredentials</i> parameter will be used as a client certificate, and the application will begin to negotiate the secure session as a client by initiating the handshake with the server. The certificate that is used must be a valid client certificate with a private key associated with it. If the <i>lpCredentials</i> parameter is NULL, then a secure client session will be initiated without a client certificate.
INET_SECURE_SERVER	The certificate specified by the <i>lpCredentials</i> parameter will be used as a server certificate, and the application will wait for the remote host to initiate the handshake that establishes the parameters of the secure session. The certificate that is used must be a valid server certificate and have a private key associated with it. The <i>lpCredentials</i> parameter cannot be NULL if this option is specified.

lpCredentials

Pointer to a SECURITYCREDENTIALS structure. This parameter may be NULL, in which case no client credentials will be provided. If client credentials are required, the structure members *dwSize*, *lpszCertStore*, and *lpszCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the SECURITYCREDENTIALS structure.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetEnableSecurity** function enables a secure communications session with the remote host, automatically negotiating the encryption algorithm and validating the certificate. This function is useful if the application needs to establish a standard connection to the remote host and then negotiate a secure connection at a later point (this is known as explicit TLS). If the function succeeds, all subsequent calls to **InetRead** and **InetWrite** to receive and send data will be encrypted.

If the *dwOptions* parameter has a value of zero and the socket was created using **InetConnect** or related functions to establish a client connection, then **InetEnableSecurity** will initiate the handshake with the remote host to establish a secure session. If the **InetAccept** or related functions were used to accept a connection from a client, then the function will block and wait for the client to initiate the handshake.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: csksv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetConnect](#), [InetConnectEx](#), [InetCreateSecurityCredentials](#), [InetDeleteSecurityCredentials](#),
[InetDisableSecurity](#)

InetEnableTrace Function

```
BOOL WINAPI InetEnableTrace(  
    LPCTSTR lpszTraceFile,  
    DWORD dwTraceFlags  
);
```

The **InetEnableTrace** function enables the logging of network function calls to a file.

Parameters

lpszTraceFile

A null-terminated string which specifies the name of the logfile which will be appended to or created. It is recommended that this file always specify an absolute path to avoid any ambiguity. The path can contain environment variable names surrounded by percent (%) symbols which will be expanded prior to opening or creating the file. If the file already exists, the current process must have read and write permission to access the file.

dwTraceFlags

An unsigned integer that specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
TRACE_INFO	All function calls are logged, but without detailed information about the data being transmitted. This is the most basic level of logging and can be useful when debugging call sequences or return values without the additional overhead of logging the data being exchanged between the client and server.
TRACE_ERROR	Only those function calls which fail are written to the file. It is important to note that this only relates to errors at the networking level and not errors which can occur at the protocol level, such as an error response from the server because the client sends an invalid request.
TRACE_WARNING	Only those function calls which fail or return values which indicate a warning are written to the file. Warnings consist of error conditions which do not normally constitute a failure at the network level. The most common warning that can occur is a timeout while waiting for a response from the remote host.
TRACE_HEXDUMP	In addition to logging function calls, this option writes the raw data that is sent and received over the network, shown in both hexadecimal and ASCII formats. This is especially useful for debugging protocol-level issues and verifying payload contents and responses.
TRACE_TRUNCATE	If the file already exists and is not read-only, the contents will be deleted before logging begins. This option is typically used to begin a new log file each time tracing is enabled, rather than appending to an existing file. This option requires the process to have permission to read, write and delete the file in the specified folder.
TRACE_PROCESS	All function calls in the current process are logged, rather than only those functions in the current thread. This option is useful for

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates an error, typically because the file could not be opened or created by the current process.

Remarks

Logging is enabled and disabled at the thread or process level, depending on the flags passed to this function, and is not associated with a specific client session. When logging is enabled, the information is appended to the specified file. If the file does not exist, it will be created. To limit the size of the file, enable and disable logging only around those sections of code that you wish to debug. If you do not provide a full path to the file, it will be relative to the current working directory for the process. This can produce unexpected results in multithreaded applications because the current directory is global to the entire process and can be changed by any thread at any time.

If logging is not enabled, there is no negative impact on performance or throughput. Once enabled, application performance can degrade, especially in those situations in which there is a large amount of data being exchanged. The log file can contain detailed information about connection attempts and authentication requests, potentially exposing user credentials, authorization tokens and API keys. If you enable logging in a production release of your application, this information can also potentially reveal file structures, API endpoints and data specific to your application. Exercise caution when preserving or sharing the contents of the file.

The TRACE_HEXDUMP option can generate very large files because it includes all of the data exchanged between your application and the remote host. The TRACE_TRUNCATE option is only evaluated when logging is first enabled for a thread or process. If logging is already active, specifying this option will have no effect.

The log file has the following format:

```
APPNAME 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9,0x27) returned 0
APPNAME 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
APPNAME 111535 0000 ERR: accept(46, NULL, 0x0) returned -1 [10038]
```

The first column contains the name of the process. The second column is the local time in hours, minutes and seconds. The third column is the elapsed time in milliseconds since the previous function call. The fourth column identifies if the logged event contains information, a warning, or an error. What follows is the name of the network function being called, the arguments passed to the function and the function's return value. If a warning or error is reported, the error code is appended to the record and the value is enclosed in brackets.

If function parameters are passed as integer values, they are recorded in decimal. If the parameter or return value is a memory address, it is recorded as a hexadecimal value preceded with "0x". Those functions which expect network addresses are displayed in one of two formats:

```
aa.bb.cc.dd:nnnn
[xxxx:xxxx:xxxx:xxxx:xxxx]:nnnn
```

With an IPv4 address, the first four numbers separated by periods represent the IP address, and the number following the colon represents the port number in host byte order. If an IPv6 address is used instead of IPv4, the address will be a series of hexadecimal digits separated by colons and enclosed in brackets. In the second line of the above example, a connection is being made to a system with the IPv4 address 192.0.0.1 on port 1234.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetDisableTrace](#)

InetEnumHostAliases Function

```
INT WINAPI InetEnumHostAliases(  
    LPCTSTR lpszHostName,  
    INT nAddressFamily,  
    LPTSTR lpszHostAliases,  
    INT nMaxLength  
);
```

The **InetEnumHostAliases** function returns a collection of null-terminated strings which contain the aliases for a specified host.

Parameters

lpszHostName

A null-terminated string which specifies a host name or IP address. This parameter cannot be NULL and must specify a valid host name.

nAddressFamily

An integer which specifies the address family which should be used when resolving the host name or IP address. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	If the host name specifies an IP address, it will be resolved based on the format of the string. For compatibility, host names are resolved to IPv4 addresses by default but if there is only an IPv6 address assigned to the host name, it will be used.
INET_ADDRESS_IPV4	Specifies the host name should be resolved using an IPv4 address.
INET_ADDRESS_IPV6	Specifies the host name should be resolved using an IPv6 address.

lpszHostAliases

A null-terminated string buffer which will contain the aliases for the specified. If this parameter is NULL, the function will return the number of characters which would be copied into the buffer.

nMaxLength

An integer value which specifies the maximum number of characters which can be copied into the buffer, including the terminating null characters. If the *lpszHostAliases* parameter is NULL, this parameter should have a value of zero.

Return Value

If the function succeeds, the return value is the number of characters copied to the string, including the terminating null characters which indicate the end of each host alias. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If there are multiple aliases for the host name, each name will be terminated with a null character, with an extra null character indicating the end of the list of aliases.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

InetEnumNetworkAddresses Function

```
INT WINAPI InetEnumNetworkAddresses(  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddressList,  
    INT nMaxAddresses  
);
```

The **InetEnumNetworkAddresses** function returns the list of network addresses that are configured for the local host.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return both IPv4 or IPv6 addresses assigned to the local host, depending on how the system is configured and which network interfaces are enabled. This option is only recommended for applications that require support for IPv6 connections.
INET_ADDRESS_IPV4	Return only the IPv4 addresses assigned to the local host. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Return only the IPv6 addresses assigned to the local host. All bytes in the <i>ipNumber</i> array are significant. This option is only recommended for those applications that require support for IPv6 connections.

lpAddressList

A pointer to an array of [INTERNET_ADDRESS](#) structures that will contain the IP address of each local network interface. This parameter may be NULL, in which case the method will only return the number of available addresses.

nMaxAddresses

Maximum number of addresses to be returned. If the *lpAddressList* parameter is NULL, this value must be zero.

Return Value

If the function succeeds, the return value is the number of network addresses that are configured for the local host. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the *nAddressFamily* parameter is specified as INET_ADDRESS_ANY, the application must be prepared to accept IPv6 addresses returned by this function. On Windows Vista and later versions of the operating system, IPv6 support is enabled and the local network adapter will have IPv6 addresses assigned to them by default. For legacy applications that only recognize IPv4 addresses, the

nAddressFamily parameter should always be specified as `INET_ADDRESS_IPV4` to ensure that only IPv4 addresses are returned.

This function will ignore addresses that are bound to a disabled interface, as well as those addresses bound to a virtual loopback interface. For example, although the loopback address 127.0.0.1 is a valid network address, it will not be included in list of addresses returned by this function.

The first IPv4 or IPv6 address returned by this function is typically the address assigned to the primary network adapter on the local system. However, your application should not depend on addresses being returned in any particular order. If the system has dial-up networking or virtualization software installed, this function may also include the IP addresses assigned to any virtualized network adapters installed by that software.

Example

```
INTERNET_ADDRESS *lpAddressList = NULL;
INT nAddressCount = InetEnumNetworkAddresses(INET_ADDRESS_IPV4, NULL, 0);

if (nAddressCount > 0)
{
    // Allocate memory for the array of IP addresses
    lpAddressList = (INTERNET_ADDRESS *)LocalAlloc(LPTR, nAddressCount *
sizeof(INTERNET_ADDRESS));

    if (lpAddressList == NULL)
    {
        // Virtual memory exhausted
        return;
    }

    // Populate the array with the addresses
    nAddressCount = InetEnumNetworkAddresses(INET_ADDRESS_IPV4, lpAddressList,
nAddressCount);
}

_tprintf(_T("There are %d local network addresses assigned\n"), nAddressCount);

// Display each IP address assigned to the local system
for (INT nIndex = 0; nIndex < nAddressCount; nIndex++)
{
    TCHAR szValue[64];

    // Convert the IP address to a printable string
    InetFormatAddress(lpAddressList + nIndex, szValue, 64);
    _tprintf(_T("%d: %s\n"), nIndex, szValue);
}

// Free the memory allocated for the IP address list
if (lpAddressList != NULL)
    LocalFree((HLOCAL)lpAddressList);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetGetAdapterAddress](#), [InetGetHostAddress](#), [InetGetLocalAddress](#)

InetEnumServerClients Function

```
INT WINAPI InetEnumServerClients(  
    SOCKET hServer,  
    SOCKET *lpClients,  
    INT nMaxClients  
);
```

The **InetEnumServerClients** function returns a list of active client sessions established with the specified server.

Parameters

hServer

Handle to the server socket.

lpClients

Pointer to an array of socket handles which identifies all client connections. If this parameter is NULL, then the function will return the number of active client connections established with the server.

nMaxClients

Maximum number of client socket handles to be returned. If the *lpClients* parameter is NULL, this parameter should be specified with a value of zero.

Return Value

If the function succeeds, the return value is the number of active client connections to the server. A return value of zero indicates that there are no active client sessions. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the *nMaxClients* parameter is less than the number of active client connections, the function will fail. To dynamically determine the number of active connections, call the function with the *lpClients* parameter with a value of NULL, and the *nMaxClients* parameter with a value of zero. To enumerate the active clients that match a specific IP address, use the **InetEnumServerClientsByAddress** function.

This function will not enumerate clients that have disconnected from the server, even if the session thread is still active. If the server is in the process of shutting down, this function will return zero, indicating no active client sessions, even though there may be clients that are still in the process of disconnecting from the server. To determine the actual number of client sessions regardless of their status, use the **InetGetClientThreads** function.

The socket handle for the server must be one that was created using the **InetServerStart** function, and cannot be a socket that was created using **InetListen** or **InetListenEx**.

Example

```
INT nMaxClients = InetEnumServerClients(hServer, NULL, 0);  
  
if (nMaxClients > 0)  
{  
    SOCKET *lpClients = NULL;  
  
    // Allocate memory for client sockets
```

```

lpClients = (SOCKET *)LocalAlloc(LPTR, nMaxClients * sizeof(SOCKET));
if (lpClients == NULL)
{
    // Virtual memory has been exhausted
    return;
}

nMaxClients = InetEnumServerClients(hServer, lpClients, nMaxClients);
if (nMaxClients == INET_ERROR)
{
    // Unable to obtain list of connected clients
    return;
}

for (INT nClient = 0; nClient < nMaxClients; nClient++)
{
    // Perform some action with each client socket
    SOCKET hClient = lpClients[nClient];
}

// Free memory allocated for client sockets
LocalFree((HLOCAL)lpClients);
}

```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetEnumServerClientsByAddress](#), [InetGetClientThreads](#), [InetServerBroadcast](#), [InetServerStart](#)

InetEventProc Function

```
VOID CALLBACK InetEventProc(  
    SOCKET hSocket,  
    UINT nEvent,  
    DWORD dwError,  
    DWORD_PTR dwParam  
);
```

The **InetEventProc** function is an application-defined callback function that processes events generated by the calling process.

Parameters

hSocket

The socket handle.

nEvent

An unsigned integer which specifies which event occurred. For a complete list of events, refer to the [InetRegisterEvent](#) function.

dwError

An unsigned integer which specifies any error that occurred. If no error occurred, then this parameter will be zero.

dwParam

A user-defined integer value which was specified when the event callback was registered.

Return Value

None.

Remarks

An application must register this callback function by passing its address to the **InetRegisterEvent** function. The **InetEventProc** function is a placeholder for the application-defined function name.

If the callback function is being used with the **InetServerStart** function, the function will be called in the context of the thread that is currently managing the server or client session. You must ensure that any access to global or static variables is synchronized, otherwise the results may be unpredictable. It is recommended that you do not declare any static variables within the callback function itself. If you need to manage state information for a specific client, then use the **InetGetClientData** and **InetSetClientData** functions which will allow you to access application defined data for that client session in a thread-safe manner.

When this callback function is used for event notifications from the server interface, the *hSocket* parameter specifies the client socket handle, except for the INET_EVENT_ACCEPT event, in which case the handle references the server socket handle. To obtain the handle of the client connection that was just accepted, use the **InetGetServerClient** function. To obtain the handle to the server using the client socket handle, use the **InetGetClientServer** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

See Also

[InetDisableEvents](#), [InetEnableEvents](#), [InetFreezeEvents](#), [InetGetClientServer](#), [InetGetServerClient](#)

InetFindClientMoniker Function

```
SOCKET WINAPI InetFindClientMoniker(  
    SOCKET hServer,  
    LPCTSTR LpszMoniker  
);
```

The **InetFindClientMoniker** function returns a handle to the client socket which matches the specified moniker.

Parameters

hServer

A handle to the server.

lpszMoniker

A null-terminated string which specifies the client moniker to search for. This parameter cannot be NULL and cannot specify an empty string.

Return Value

If the function succeeds, the return value is the handle to the client socket for the session that matches the specified moniker. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session using the **InetSetClientMoniker** function. This function will search all active client sessions for the server, and returns the socket handle to the client that matches the specified moniker. If there is no match, an error will be returned.

The moniker can be any string value, however monikers are not case sensitive and may not contain embedded null characters. The maximum length of a moniker is 127 characters.

The socket handle for the server must be one that was created using the **InetServerStart** function, and cannot be a socket that was created using **InetListen** or **InetListenEx**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetClientMoniker](#), [InetGetClientServer](#), [InetGetClientThreadId](#), [InetSetClientMoniker](#)

InetFlush Function

```
INT WINAPI InetFlush(  
    SOCKET hSocket  
);
```

The **InetFlush** function flushes the internal send and receive buffers used by the socket.

Parameters

hSocket

The socket handle.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetFlush** function will flush any data waiting to be read or written to the remote host . It is important to note that this function is not similar to flushing data to a disk file; it does not ensure that a specific block of data has been written to the socket. For example, you should never call this function immediately after calling **InetWrite** or prior to calling **InetDisconnect**.

An application never needs to use **InetFlush** under normal circumstances. This function is only to be used when the application needs to immediately return the socket to an inactive state with no pending data to be read or written. Calling this function may result in data loss and should only be used if you understand the implications of discarding any data which has been sent by the remote host.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetIsReadable](#), [InetIsWritable](#), [InetPeek](#), [InetRead](#), [InetWrite](#)

InetFormatAddress Function

```
INT WINAPI InetFormatAddress(  
    LPINTERNET_ADDRESS LpAddress,  
    LPTSTR LpszAddress,  
    INT cchAddress  
);
```

The **InetFormatAddress** function converts a numeric IP address to a printable string. The format of the string depends on whether an IPv4 or IPv6 address is specified.

Parameters

LpAddress

A pointer to an [INTERNET_ADDRESS](#) structure which specifies the numeric IP address that should be converted to a string.

LpszAddress

A null-terminated string buffer that will contain the formatted IP address, terminated with a null character. To accommodate both IPv4 and IPv6 addresses, this buffer should be at least 46 characters in length.

cchAddress

The maximum number of characters that can be copied into the address buffer.

Return Value

If the function succeeds, the return value is the length of the IP address string. If the function fails, the return value is `INET_ERROR`, meaning that the IP address could not be converted into a string. Typically this indicates that the pointer to the `INTERNET_ADDRESS` structure is invalid, or the data does not specify a valid IP address family.

Remarks

The format and length of IPv4 and IPv6 address strings are very different. An IPv4 address string looks like "192.168.0.20", while an IPv6 address string can look something like "fd7c:2f6a:4f4f:ba34::a32". If your application checks for the format of these address strings, it needs to be aware of the differences. You also need to make sure that you're providing enough space to display or store an address to avoid buffer overruns.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetExternalAddress](#), [InetGetHostAddress](#), [InetGetLocalAddress](#), [InetGetPeerAddress](#), [INTERNET_ADDRESS](#)

InetFreezeEvents Function

```
INT WINAPI InetFreezeEvents(  
    SOCKET hSocket,  
    BOOL bFreeze  
);
```

The **InetFreezeEvents** function is used to suspend and resume event handling.

Parameters

hSocket

Socket handle.

bFreeze

A non-zero value specifies that event handling should be suspended. A zero value specifies that event handling should be resumed.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

This function should be used when the application does not want to process events, such as when a modal dialog is being displayed. When events are suspended, all events are queued. If events are re-enabled at a later point, those queued events will be sent to the application for processing. Note that only one of each event will be generated. For example, if event handling has been suspended, and four read events occur, only one of those read events will be posted to the application when even handling is resumed. This prevents the application from being flooded by a potentially large number of queued events.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetDisableEvents](#), [InetEnableEvents](#), [InetRegisterEvent](#)

InetGetAdapterAddress Function

```
INT WINAPI InetGetAdapterAddress(  
    INT nAdapterIndex,  
    INT nAddressType,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

Return the IP or MAC assigned to the specified network adapter.

Parameters

nAdapterIndex

An integer value that identifies the network adapter.

nAddressType

An integer value which specifies the type of address that should be returned:

Value	Description
INET_ADAPTER_IPV4	The address string will contain the primary IPv4 unicast address assigned to the network adapter.
INET_ADAPTER_IPV6	The address string will contain the primary IPv6 unicast address assigned to the network adapter.
INET_ADAPTER_MAC	The address string will contain the media access control (MAC) address assigned to the network adapter.

lpszAddress

A string buffer that will contain the IP or MAC address assigned to the adapter. This parameter cannot be NULL and it is recommended that it be at least 64 characters in length to provide enough space for any address type.

nMaxLength

The maximum number of characters that can be copied into the string buffer, including the terminating null character. If the buffer is too small to store the complete address, this function will fail.

Return Value

If the function succeeds, the return value is the number of characters copied to the string buffer, not including the terminating null character. A return value of zero indicates that the requested address type has not been assigned to the adapter. If the function fails, the return value is INET_ERROR and this typically indicates that either the adapter index is invalid or the string buffer is not large enough to store the complete address. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetAdapterAddress** function will return the IPv4, IPv6 or MAC address assigned to a specific network adapter. The primary network adapter has an index value of zero, and it increments for each adapter that is configured on the local system.

The media access control (MAC) address is a 48 bit or 64 bit value that is assigned to each network interface and is used for identification and access control. All network devices on the same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned

dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This function returns the MAC address string as sequence of hexadecimal values separated by a colon. An example of a 48 bit MAC address would be "01:23:45:67:89:AB". Note that some virtual network adapters may not have a MAC address assigned to them, in which case this function would return zero.

This function will ignore network adapters that have been disabled, as well as those that are bound to a virtual loopback interface. If the system has dial-up networking or virtualization software installed, this function may also return IP addresses assigned to a virtualized network adapters installed by that software.

Example

```
// Display the IPv4 address assigned to each network adapter
for (INT nIndex = 0;; nIndex++)
{
    TCHAR szAddress[64];
    INT cchAddress;

    cchAddress = InetGetAdapterAddress(nIndex, INET_ADAPTER_IPV4, szAddress, 64);

    if (cchAddress == INET_ERROR)
        break;

    _tprintf(_T("Adapter %d: %s\n"), nIndex, szAddress);
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetEnumNetworkAddresses](#), [InetGetLocalAddress](#), [InetGetLocalName](#)

InetGetAddress Function

```
INT WINAPI InetGetAddress(  
    LPCTSTR lpszAddress,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **InetGetAddress** function converts an IP address string to binary format.

Parameters

lpszAddress

A null-terminated string which specifies an IP address. This function recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address specified by the *lpszAddress* parameter. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the value of the <i>lpszAddress</i> parameter.
INET_ADDRESS_IPV4	Specifies that the address should be in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero. If the <i>lpszAddress</i> parameter does not specify a valid IPv4 address string, this function will fail.
INET_ADDRESS_IPV6	Specifies that the address should be in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. If the <i>lpszAddress</i> parameter does not specify a valid IPv6 address string, this function will fail.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

If the *nAddressFamily* parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible that an IPv6 address string has been specified. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned and any attempt to specify an IPv6 address string would result in an error.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system,

use the **InetIsProtocolAvailable** function. If an IPv6 stack is not installed, this function will fail if the *lpzAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetFormatAddress](#), [InetIsAddressNull](#), [InetIsAddressRoutable](#), [InetIsProtocolAvailable](#),
[INTERNET_ADDRESS](#)

InetGetBlockingSocket Function

```
SOCKET InetGetBlockingSocket();
```

The **InetGetBlockingSocket** function returns the handle for the socket in the current thread which is currently blocking, if there is one.

Parameters

None.

Return Value

If the function succeeds, the return value is the handle for the socket in the current thread which is currently blocking. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cssock12.h`

Import Library: `cswskv12.lib`

See Also

[InetAbort](#), [InetCancel](#)

InetGetClientData Function

```
BOOL WINAPI InetGetClientData(  
    SOCKET hClient,  
    LPVOID *lppvData  
);
```

The **InetGetClientData** function returns the application defined data associated with the specified client session.

Parameters

hClient

The client socket handle.

lppvData

Pointer to a void pointer which will contain an application defined value associated with the client session.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the client session could not be retrieved. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientData** function is used to retrieve the application defined data that was previously associated with a client session using the **InetSetClientData** function. This is typically used to associate a pointer to a data structure or a class instance with a specific client handle.

This function can only be used with client socket handles created using the server interface. It cannot be used with socket handles created using the **InetConnect** or **InetAccept** functions. If the socket handle is invalid, or does not reference a client socket handle created by the server, the *lppvData* pointer passed to this function will be initialized to a value of NULL and the function will return a value of zero.

If this function is called with a valid socket handle and there is no data associated with the socket, the function will return a non-zero value and the *lppvData* pointer will be returned with a NULL value. Before dereferencing the pointer returned by this function, the application should always check the return value to ensure the function succeeded and make sure that the pointer is not NULL.

Example

```
UINT *pnValue1 = (UINT *)LocalAlloc(LPTR, sizeof(UINT));  
UINT *pnValue2 = NULL;  
  
*pnValue1 = 1234;  
  
if (InetSetClientData(hClient, pnValue1) == FALSE)  
{  
    // Unable to associate the data with this session  
    return;  
}  
  
if (InetGetClientData(hClient, &pnValue2) == FALSE)  
{
```

```
    // Unable to retrieve the data associated with this session
    return;
}

// *pnValue2 == 1234
printf("The value of user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetServerData](#), [InetSetClientData](#), [InetSetServerData](#)

InetGetClientHandle Function

```
SOCKET WINAPI InetGetClientHandle(  
    SOCKET hServer,  
    UINT nClientId  
);
```

The **InetGetClientHandle** function returns the handle for a specific client session based on its ID number.

Parameters

hServer

Handle to the server socket.

nClientId

An unsigned integer value which uniquely identifies the client session.

Return Value

If the function succeeds, the return value is the socket handle for the specified client session. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call

InetGetLastError.

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value. This value can be obtained by calling the **InetGetClientId** function and used by the application to identify that client session. The **InetGetClientHandle** function can then be used to obtain the client socket handle for the session, based on that client ID.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetClientId](#), [InetGetClientMoniker](#), [InetSetClientMoniker](#), [InetGetServerClient](#), [InetGetThreadClient](#)

InetGetClientId Function

```
UINT WINAPI InetGetClientId(  
    SOCKET hClient  
);
```

The **InetGetClientId** function returns the unique ID number assigned to the specified client session.

Parameters

hClient

Handle to the client socket.

Return Value

If the function succeeds, the return value is an unsigned integer value which uniquely identifies the client session. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value. This value can be obtained by calling the **InetGetClientId** function and used by the application to identify that client session. The **InetGetClientHandle** function can then be used to obtain the client socket handle for the session, based on that client ID. It is important to note that the actual value of the client ID should be considered opaque. It is only guaranteed that the value will be greater than zero, and that it will be unique to the client session.

While it is possible for a client socket handle to be reused by the operating system, client IDs are unique throughout the life of the server session and are never duplicated.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetClientHandle](#), [InetGetClientMoniker](#), [InetSetClientMoniker](#), [InetGetServerClient](#)

InetGetClientIdleTime Function

```
DWORD WINAPI InetGetClientIdleTime(  
    SOCKET hClient  
);
```

Returns the number of milliseconds that the specified client session has been idle.

Parameters

hClient

Handle to the client socket.

Return Value

If the function succeeds, the return value is an unsigned integer value which specifies the number of milliseconds the client session has been idle. If the function fails, the return value is INFINITE. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientIdleTime** function will return the number of milliseconds that have elapsed since data was exchanged with the client. The elapsed time is limited to the resolution of the system timer, which is typically in the range of 10 milliseconds to 16 milliseconds.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

See Also

[InetGetClientHandle](#), [InetGetClientMoniker](#), [InetGetServerClient](#)

InetGetClientMoniker Function

```
INT WINAPI InetGetClientMoniker(  
    SOCKET hSocket,  
    LPTSTR lpszMoniker,  
    INT nMaxLength  
);
```

The **InetGetClientMoniker** function returns the moniker associated with the specified client session.

Parameters

hSocket

Handle to the client socket.

lpszMoniker

Pointer to a string buffer that will contain the moniker for the specified client session when the function returns.

nMaxLength

The maximum number of characters that may be copied into the string buffer. The buffer must be large enough to store the moniker and a terminating null character. The maximum length of a moniker is 127 characters.

Return Value

If the function succeeds, the return value is the number of characters in the moniker string. A return value of zero specifies that no moniker was assigned to the socket. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. A moniker can be assigned to the client session using the **InetSetClientMoniker** function. This function will return the moniker that was previously assigned to the client, if any. To obtain the socket handle associated with a given moniker, use the **InetFindClientMoniker** function.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetFindClientMoniker](#), [InetGetClientHandle](#), [InetGetClientId](#), [InetSetClientMoniker](#)

InetGetClientPriority Function

```
INT WINAPI InetGetClientPriority(  
    SOCKET hClient  
);
```

The **InetGetClientPriority** function returns the current priority for the specified client session.

Parameters

hClient
Handle to the client socket.

Return Value

If the function succeeds, the return value is the priority for the specified client session. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientPriority** function can be used to determine the current priority assigned to the specified client session. The client priority is inherited from the priority specified when the server is started using the **InetServerStart** function. It may be one of the following values:

Value	Description
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. It is recommended that most applications use this priority.
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the client session. It is typically used with lightweight services running in the background that are designed for few client connections. The client thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the client session and meters the processor utilization for the client session. The client thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the client session and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The client thread will be given higher scheduling priority and will be more responsive to network events. It is not recommended that this priority be used on a system with a single processor.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

See Also

[InetGetServerPriority](#), [InetServerStart](#), [InetSetClientPriority](#), [InetSetServerPriority](#)

InetGetClientServer Function

```
SOCKET WINAPI InetGetClientServer(  
    SOCKET hClient  
);
```

The **InetGetClientServer** function returns a socket handle to the server for the specified client socket.

Parameters

hClient

Handle to the client socket.

Return Value

If the function succeeds, the return value is the handle to the server that created the client session. If the function fails, the return value is `INVALID_SOCKET`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientServer** function returns the handle to the server that created the client session. The **InetGetClientServerById** function can be used to obtain the server handle using the client session ID rather than the client socket handle.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetGetClientHandle](#), [InetGetClientId](#), [InetGetClientServerById](#)

InetGetClientServerById Function

```
SOCKET WINAPI InetGetClientServerById(  
    UINT nClientId  
);
```

The **InetGetClientServerById** function returns a socket handle to the server for the specified client session identifier.

Parameters

nClientId

Client session identifier.

Return Value

If the function succeeds, the return value is the handle to the server that created the client session. If the function fails, the return value is `INVALID_SOCKET`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientServerById** function returns the handle to the server that created the client session using the client's unique identifier. The **InetGetClientServer** function can be used to obtain the server handle using the client socket handle rather than the client session ID. This function is typically used in conjunction with the `INET_NOTIFY_CONNECT` notification message to obtain the handle to the server that generated the event using the client ID passed in the *wParam* message parameter.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetGetClientHandle](#), [InetGetClientId](#), [InetGetClientServer](#), [InetServerAsyncNotify](#)

InetGetClientThreadId Function

```
DWORD WINAPI InetGetClientThreadId(  
    SOCKET hClient  
);
```

The **InetGetClientThreadId** function returns the thread ID for the specified client session.

Parameters

hClient

Handle to the client socket.

Return Value

If the function succeeds, the return value is an unsigned integer value which identifies the thread that was created to manage the client session. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The thread ID returned by this function can be used with the **OpenThread** function to obtain a handle to the thread. Until the thread terminates, the thread identifier uniquely identifies the thread throughout the system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetClientPriority](#), [InetGetServerThreadId](#), [InetGetThreadClient](#), [InetSetClientPriority](#)

InetGetClientThreads Function

```
INT WINAPI InetGetClientThreads(  
    SOCKET hServer  
);
```

Returns the number of client session threads created by the server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the number of client session threads that have been created by the server. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetClientThreads** function returns the number of threads that are managing client sessions for the specified server. If there are no clients connected to the server, this function will return a value of zero. Because this function returns the number of session threads, the value returned will include those clients that are in the process of disconnecting from the server but their session thread has not yet terminated. This differs from the **InetEnumServerClients** function which will only enumerate active clients.

If you wish to determine when the last client has disconnected from the server, call this function within an event handler for the INET_EVENT_DISCONNECT event. If the function returns a value greater than one, then there are other client sessions that are either connected or in the process of terminating.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

See Also

[InetEnumServerClients](#), [InetEnumServerClientsByAddress](#)

InetGetDefaultHostFile Function

```
INT WINAPI InetGetDefaultHostFile(  
    LPTSTR lpszFileName,  
    INT nMaxLength  
);
```

The **InetGetDefaultHostFile** function returns the fully qualified path name of the host file on the local system. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **InetGetHostAddress** and **InetGetHostNames** function. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

The location of the default host file depends on the operating system. For Windows 95/98 and Windows Me the file is stored in C:\Windows\hosts and for Windows NT and later versions the file is stored in C:\Windows\system32\drivers\etc\hosts. Regardless of platform, there is no filename extension and this file may or may not exist on a given system.

Parameters

lpszFileName

Pointer to a string buffer that will contain the fully qualified file name to the default host file. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the function will return the length of the string, not including the terminating null byte.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the function succeeds, the return value is length of the string. A return value of zero indicates that the default host file could not be determined for the current platform. To get extended error information, call **InetGetLastError**.

Remarks

This function only returns the default location of the host file and does not determine if the file actually exists. It is not required that a host file be present on the system.

The default host file is processed before performing a nameserver lookup when resolving a hostname into an IP address, or an IP address into a hostname.

To specify an alternate local host file, use the **InetSetHostFile** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetHostAddress](#), [InetGetHostFile](#), [InetGetHostName](#), [InetSetHostFile](#)

InetGetErrorString Function

```
INT WINAPI InetGetErrorString(  
    DWORD dwErrorCode,  
    LPTSTR lpszDescription,  
    INT nMaxLength  
);
```

The **InetGetErrorString** function is used to return a description of a specific error code. Typically this is used in conjunction with the [InetGetLastError](#) function for use with warning dialogs or as diagnostic messages.

Parameters

dwErrorCode

The last-error code for which a description is returned.

lpszDescription

The buffer that will contain a description of the specified error code. This buffer should be at least 128 characters in length.

nMaxLength

The maximum number of characters that may be copied into the description buffer.

Return Value

If the function succeeds, the return value is the length of the description string. If the function fails, the return value is 0, meaning that no description exists for the specified error code. Typically this indicates that the error code passed to the function is invalid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetLastError](#), [InetSetLastError](#)

InetGetExternalAddress Function

```
INT WINAPI InetGetExternalAddress(  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **InetGetExternalAddress** function returns the external IP address for the local system.

Parameters

nAddressFamily

An integer which identifies the type of IP address that should be returned by this function. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return any external IP address available. IPv4 addresses will be preferred over IPv6 for backwards compatibility.
INET_ADDRESS_IPV4	Specifies that the external address should be in IPv4 format.
INET_ADDRESS_IPV6	Specifies that the external address should be in IPv6 format. If the local system does not have IPv6 enabled, the function will fail.

lpAddress

A pointer to an INTERNET_ADDRESS structure which contains the external IP address of the local system when the function returns.

Return Value

If the function succeeds, the return value is zero and the INTERNET_ADDRESS structure contains the external IP address for the local host in binary form. If the function fails, it will return INET_ERROR and the *ipFamily* member of the INTERNET_ADDRESS structure will be set to INET_ADDRESS_UNKNOWN. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetExternalAddress** function returns the IP address assigned to the router that connects the local host to the Internet. This is typically used by an application executing on a system in a local network that uses a router which performs Network Address Translation (NAT). In that network configuration, the **InetGetLocalAddress** function will only return the IP address for the local system on the LAN side of the network unless a connection has already been established to a remote host. The **InetGetExternalAddress** function can be used to determine the IP address assigned to the router on the Internet side of the connection and can be particularly useful for servers running on a system behind a NAT router.

An application should always check the return value in case there is an error. Never assume that the return value is always a valid address. The function may be unable to determine the external IP address for the local system for a number of reasons, particularly if it is behind a firewall or uses a proxy server that restricts access to external sites on the Internet. If the function is able to obtain a valid external address for the local host, that address will be cached by the library. If you are using a VPN (Virtual Private Network), this function will return the external IP address for the VPN adapter and not your local network connection. The **InetGetVpnAdapter** function can be used to obtain information about the active VPN adapter.

Calling this function may cause the current thread to block until the external IP address can be resolved and should never be used in conjunction with asynchronous socket connections. If you need to call this function in an application which uses asynchronous sockets, it is recommended that you create a background worker thread and call this function from within that thread.

To convert the address from its binary form into a string, use the **InetFormatAddress** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: csoskv12.lib

See Also

[InetFormatAddress](#), [InetGetHostAddress](#), [InetGetLocalAddress](#), [InetGetPeerAddress](#),
[InetGetVpnAdapter](#), [INTERNET_ADDRESS](#)

InetGetHostAddress Function

```
INT WINAPI InetGetHostAddress(  
    LPCTSTR lpszHostName,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **InetGetHostAddress** function resolves the specified host name into an IP address in binary format.

Parameters

lpszHostName

The name of the host to resolve; this may be a fully-qualified domain name or an IP address. This function recognizes the format for both IPv4 and IPv6 format addresses.

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on how the host name can be resolved. By default, a preference will be given for returning an IPv4 address. However, if the host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an INTERNET_ADDRESS structure that will contain the IP address of the specified host.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

This function can also be used to convert an address in dot notation to a binary format. If the function must perform a DNS lookup to resolve the hostname, the calling thread will block. To ensure future compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the *ipFamily* member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

The *nAddressFamily* parameter is used to specify a preference for the type of address returned,

however it is possible that a host may not have an IPv4 or IPv6 address record, in which case this function will fail. Although IPv4 is still the most common address used at this time, an application should not assume that because a given host name does not have an IPv4 address, that the host name is invalid.

If the *nAddressFamily* parameter is specified as INET_ADDRESS_UNKNOWN, the application must be prepared to handle IPv6 addresses because it is possible for a host name to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as INET_ADDRESS_IPV4 to ensure that only IPv4 addresses are returned.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **InetIsProtocolAvailable** function. If an IPv6 stack is not installed, this function will fail if the *lpzHostName* parameter specifies an host that only has an IPv6 (AAAA) DNS record.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetHostName](#), [InetGetLocalAddress](#), [InetGetLocalName](#), [InetGetPeerAddress](#), [InetIsProtocolAvailable](#), [INTERNET_ADDRESS](#)

InetGetHostFile Function

```
INT WINAPI InetGetHostFile(  
    LPTSTR lpzFileName,  
    INT nMaxLength  
);
```

The **InetGetHostFile** function returns the name of the host file previously set using the **InetSetHostFile** function. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **InetGetHostAddress** and **InetGetHostNames** function.

Parameters

lpzFileName

Pointer to a string buffer that will contain the host file name. It is recommended that this buffer be at least MAX_PATH characters in size. This parameter may be NULL, in which case the function will return the length of the string, not including the terminating null character.

nMaxLength

The maximum number of characters that may be copied to the string buffer.

Return Value

If the function succeeds, the return value is length of the string. A return value of zero indicates that no host file has been specified or the function was unable to determine the file name. To get extended error information, call **InetGetLastError**. If the last error is zero, this indicates that no host file name has been specified for the current thread. If the last error is non-zero, this indicates the reason that the function failed.

Remarks

This function only returns the name of the host file that is cached in memory for the current thread. The contents of the file on the disk may have changed after the file was loaded into memory. To reload the host file or clear the cache, call the **InetSetHostFile** function.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

The host file returned by this function may be different than the default host file for the local system. To determine the file name for the default host file, use the **InetGetDefaultHostFile** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cssock12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetDefaultHostFile](#), [InetGetHostAddress](#), [InetGetHostName](#), [InetSetHostFile](#)

InetGetHostName Function

```
INT WINAPI InetGetHostName(  
    LPINTERNET_ADDRESS LpAddress,  
    LPTSTR LpszHostName,  
    INT cchHostName  
);
```

The **InetGetHostName** function performs a reverse lookup, returning the host name associated with a given IP address.

Parameters

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure which specifies the IP address that should be resolved into a host name.

lpszHostName

The buffer that will contain the host name. It is recommended that this buffer be at least 256 characters in length to accommodate the longest possible fully qualified domain name.

cchHostName

The maximum number of characters that can be copied into the buffer.

Return Value

If the function succeeds, the return value is the length of the hostname. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

If the function must perform a reverse DNS lookup to resolve the IP address into a host name, the calling thread will block. This function requires that the host have a PTR record, otherwise it will fail. Because many hosts do not have a PTR record, calling this function frequently may have a negative impact on the overall performance of the application.

To determine if the local system has an IPv6 TCP/IP stack installed and configured on the local system, use the **InetIsProtocolAvailable** function. If an IPv6 stack is not installed, this function will fail if the *lpAddress* parameter specifies an IPv6 address, even if the address itself is valid.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetHostAddress](#), [InetGetLocalAddress](#), [InetGetLocalName](#), [InetGetPeerAddress](#),
[InetGetUrlHostName](#), [InetIsProtocolAvailable](#), [INTERNET_ADDRESS](#)

InetGetLastError Function

DWORD WINAPI InetGetLastError();

Parameters

None.

Return Value

Functions set this value by calling the [InetSetLastError](#) function. The return value section of each reference page notes the conditions under which the function sets the last-error code.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. You should call the **InetGetLastError** function immediately when a function's return value indicates that an error has occurred. That is because some functions call **InetSetLastError**(0) when they succeed, clearing the error code set by the most recently failed function.

Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those functions which call **InetSetLastError** when they succeed are noted on the function reference page.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cwssock12.h

Import Library: cswskv12.lib

See Also

[InetGetErrorString](#), [InetSetLastError](#)

InetGetLocalAddress Function

```
INT WINAPI InetGetLocalAddress(  
    SOCKET hSocket,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS LpAddress,  
    UINT *LpnPort  
);
```

The **InetGetLocalAddress** function returns the local IP address and port number for the specified socket.

Parameters

hSocket

The socket handle.

nAddressFamily

An integer which identifies the type of IP address to return. It may be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	Return the IP address for the specified host in either IPv4 or IPv6 format, depending on the type of connection that was established. If the <i>hSocket</i> parameter is INVALID_SOCKET, a preference will be given for returning an IPv4 address. However, if the local host only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the address should be returned in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address should be returned in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the local host. If the *hSocket* parameter is specified as INVALID_SOCKET, this function will attempt to determine the IP address of the local host assigned by the system. If the address is not required, this parameter may be NULL.

lpnPort

A pointer to an unsigned integer that will contain the local port number. If the *hSocket* parameter specifies a valid socket, this parameter will be set to the local port that the socket was bound to. If the *hSocket* parameter is specified as INVALID_SOCKET, this parameter is ignored. If the port number is not required, this parameter may be NULL.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

To ensure future compatibility with IPv6 networks, it is important that the application does not make any assumptions about the format of the address. If the function returns successfully, the *ipFamily* member of the **INTERNET_ADDRESS** structure should always be checked to determine the type of address.

If the *nAddressFamily* parameter is specified as `INET_ADDRESS_UNKNOWN`, the application must be prepared to handle IPv6 addresses because it is possible for the local host to have an IPv6 address assigned to it and no IPv4 address. For legacy applications that only recognize IPv4 addresses, the *nAddressFamily* member should always be specified as `INET_ADDRESS_IPV4` to ensure that only IPv4 addresses are returned.

If the system is connected to the Internet through a local network and/or uses a router that performs Network Address Translation (NAT), the **InetGetLocalAddress** function will return the local, non-routable IP address assigned to the system. To determine the public IP address has been assigned to the system, you should use the **InetGetExternalAddress** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetGetExternalAddress](#), [InetGetHostAddress](#), [InetGetHostName](#), [InetGetLocalName](#), [InetGetPeerAddress](#), [INTERNET_ADDRESS](#)

InetGetLocalName Function

```
INT WINAPI InetGetLocalName(  
    LPTSTR LpszHostName,  
    INT cchHostName  
);
```

The **InetGetLocalName** function returns the hostname assigned to the local system.

Parameters

LpszHostName

The buffer that will contain the hostname.

cchHostName

The maximum number of characters that can be copied into the address buffer.

Return Value

If the function succeeds, the return value is the length of the hostname. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetHostAddress](#), [InetGetHostName](#), [InetGetLocalAddress](#), [InetGetPeerAddress](#)

InetGetLockedServer Function

```
SOCKET WINAPI InetGetLockedServer(  
    LPDWORD LpdwThreadId  
);
```

The **InetGetLockedServer** function unlocks the specified server, allowing other server threads to resume execution.

Parameters

LpdwThreadId

A pointer to an unsigned integer which identifies the thread that established the server lock. If this information is not required, this parameter may be NULL.

Return Value

If the function succeeds, the return value is the handle to the locked server. If no server is in a locked state, the function will return a value of INVALID_SOCKET.

Remarks

The **InetGetLockedServer** function can be used to determine if there is a server in a locked state. If there is, the function will return a handle to the server and will identify the thread which established the lock. This function may be called from any thread, however only the thread which established the server lock may interact with the server or release the lock.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetServerLock](#), [InetServerUnlock](#)

InetGetMemoryUsage Function

```
BOOL WINAPI InetGetMemoryUsage(  
    SOCKET hSocket,  
    ULARGE_INTEGER *LpMemUsage  
);
```

Return the amount of memory allocated for the network connection.

Parameters

hSocket

The socket handle.

lpMemUsage

A pointer to a ULARGE_INTEGER variable which will specify how much memory has been allocated for the socket. This parameter will be initialized to a value of zero by the function and updated with the total number of bytes allocated.

Return Value

If the function succeeds, the return value is non-zero and the memory usage value will be updated. If the socket handle does not specific a socket created by the library, the return value is zero. Call the **InetGetLastError** function to determine the cause of the failure.

Remarks

This function returns the amount of memory allocated for the socket. It enumerates all heap allocations made by the process and returns the total number of bytes allocated for the specific socket handle. If the *hSocket* parameter value is INVALID_SOCKET, the function will calculate the amount of memory allocated for all sockets created by the library. This value only reflects the amount of memory explicitly allocated by this library and does not reflect the total working set size of the process, or memory allocated by any other libraries. To determine the working set size for the process, refer to the Win32 **GetProcessWorkingSetSize** and **GetProcessMemoryInfo** functions.

This function forces the current thread into a locked state, and all network operations will block until the function returns. Because this function enumerates all heaps created for the process, it can be an expensive operation, particularly when a large number of connections have been established. Frequent use of this function can significantly degrade the performance of the application. It is primarily intended for use as a debugging tool to determine if memory usage is the result of an increase in active network connections. If the value returned by the function remains reasonably constant, but the amount of memory allocated for the process continues to grow, it could indicate a memory leak in some other area of the application.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetServerStackSize](#), [InetSetServerStackSize](#)

InetGetOption Function

```
INT WINAPI InetGetOption(  
    SOCKET hSocket,  
    DWORD dwOption,  
    LPBOOL lpbEnabled  
);
```

The **InetGetOption** function is used to determine if a specific socket option has been enabled.

Parameters

hSocket

The socket handle.

dwOption

An unsigned integer used to specify one of the socket options. These options cannot be combined. The following values are recognized:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_KEEPAIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.

lpbEnabled

A pointer to a boolean flag. If the option is enabled, the flag is set to a non-zero value, otherwise it is set to a value of zero.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Requirements

- Minimum Desktop Platform:** Windows 7 Service Pack 1
- Minimum Server Platform:** Windows Server 2008 R2 Service Pack 1
- Header File:** cswsock12.h
- Import Library:** cswskv12.lib

See Also

[InetAsyncConnectEx](#), [InetConnectEx](#), [InetSetOption](#)

InetGetPeerAddress Function

```
INT WINAPI InetGetPeerAddress(  
    SOCKET hSocket,  
    LPINTERNET_ADDRESS LpAddress,  
    UINT *LpnPort  
);
```

The **InetGetPeerAddress** function returns the peer IP address and remote port number for the specified socket.

Parameters

hSocket

The socket handle.

lpAddress

A pointer to an [INTERNET_ADDRESS](#) structure that will contain the IP address of the remote host that the socket is connected to.

lpnPort

A pointer to an unsigned integer that will contain the remote port that the socket is connected to.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

If this function is called by a server application in response to a `INET_EVENT_ACCEPT` event, it will return the IP address and port number for the client that is attempting to establish the connection. If the peer address is unavailable, the *ipFamily* member of the `INTERNET_ADDRESS` structure will be zero. To convert the IP address to a printable string, use the **InetFormatAddress** function.

It is not recommended that you use the port number for anything other than informational and logging purposes. Server applications should not make any assumptions about the specific port number or range of port numbers that a client is using when establishing a connection to the server. The ephemeral port number that a client is bound to can vary based on the client operating system.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetFormatAddress](#), [InetGetHostAddress](#), [InetGetHostName](#), [InetGetLocalAddress](#), [InetGetLocalName](#), [INTERNET_ADDRESS](#)

InetGetPhysicalAddress Function

```
BOOL WINAPI InetGetPhysicalAddress(  
    LPTSTR lpszAddress,  
    UINT nMaxLength  
);
```

Return the media access control (MAC) address for the primary network adapter.

Parameters

lpszAddress

A string buffer that will contain the address in a printable format when the function returns. This parameter cannot be NULL.

nMaxLength

The maximum number of characters that can be copied into the buffer, including the terminating null character.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetPhysicalAddress** function returns the media access control (MAC) address for the primary network adapter. This is a 48 bit or 64 bit address that is assigned to each network interface and is used for identification and access control. All network devices on the same subnet must be assigned their own unique MAC address. Unlike IP addresses which may be assigned dynamically and can be frequently changed, MAC addresses are considered to be more permanent because they are usually assigned by the device manufacturer and stored in firmware. Note that in some cases it is possible to change the address assigned to a device, and virtual network interfaces may have configurable MAC addresses.

This function returns the MAC address as a printable string, with each byte of the address as a two-digit hexadecimal value separated by a colon. The string buffer passed to the function should be at least 20 characters long to accommodate the address and terminating null character. An example of a 48 bit address would be "01:23:45:67:89:AB". If the local system is multi-homed (having more than one network adapter) then this function will return the MAC address for the primary network adapter.

This function is provided for backwards compatibility with previous versions of the library and it is recommended that new applications use the **InetGetAdapterAddress** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetEnumNetworkAddresses](#), [InetGetAdapterAddress](#), [InetGetLocalName](#), [InetGetHostAddress](#)

InetGetSecurityInformation Function

```
BOOL WINAPI InetGetSecurityInformation(  
    SOCKET hSocket,  
    LPSECURITYINFO lpSecurityInfo  
);
```

The **InetGetSecurityInformation** function fills a structure with information about the security characteristics of a connection.

Parameters

hSocket

Handle to the socket.

lpSecurityInfo

A pointer to a [SECURITYINFO](#) structure which contains information about the current client connection. The *dwSize* member of this structure must be initialized to the size of the structure before passing the address of the structure to this function.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

This function is used to obtain security related information about the current client connection to the server. It can be used to determine if a secure connection has been established, what security protocol was selected, and information about the server certificate. Note that a secure connection has not been established, the *dwProtocol* structure member will contain the value `SECURITY_PROTOCOL_NONE`.

Example

The following example notifies the user if the connection is secure or not:

```
SECURITYINFO securityInfo;
```

```
securityInfo.dwSize = sizeof(SECURITYINFO);  
if (InetGetSecurityInformation(hClient, &securityInfo))  
{  
    if (securityInfo.dwProtocol == SECURITY_PROTOCOL_NONE)  
    {  
        MessageBox(NULL, _T("The connection is not secure"),  
                    _T("Connection"), MB_OK);  
    }  
    else  
    {  
        if (securityInfo.dwCertStatus == SECURITY_CERTIFICATE_VALID)  
        {  
            MessageBox(NULL, _T("The connection is secure"),  
                        _T("Connection"), MB_OK);  
        }  
        else  
        {  
            MessageBox(NULL, _T("The server certificate not valid"),  
                        _T("Connection"), MB_OK);  
        }  
    }  
}
```

```
}  
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetConnectEx](#), [InetAsyncConnectEx](#)

InetGetServerClient Function

```
SOCKET WINAPI InetGetServerClient(  
    SOCKET hServer  
);
```

The **InetGetServerClient** function returns the handle for the last client connection accepted by the server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the handle to the last client connection that was accepted by the server. If the function fails, the return value is `INVALID_SOCKET`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerClient** function can be used inside the service event handler to determine the client connection that was just accepted by the server. This would typically be used in conjunction with the `INET_EVENT_ACCEPT` handler, enabling the application to obtain the handle of the new client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetGetClientThreadId](#), [InetGetServerThreadId](#), [InetServerBroadcast](#), [InetServerLock](#), [InetServerStop](#), [InetServerThrottle](#), [InetServerUnlock](#)

InetGetServerData Function

```
BOOL WINAPI InetGetServerData(  
    SOCKET hServer,  
    LPVOID *lppvData  
);
```

The **InetGetServerData** function returns the application defined data associated with the specified server.

Parameters

hServer

The server socket handle.

lppvData

Pointer to a void pointer which will contain an application defined value associated with the server.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the server could not be retrieved. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerData** function is used to retrieve the application defined data that was previously associated with a server using the **InetSetServerData** function. This is typically used to associate a pointer to a data structure or a class instance with a specific instance of a server.

This function can only be used with socket handles created using **InetServerStart** function. It cannot be used with socket handles created using the **InetListen** or **InetListenEx** functions. If the socket handle is invalid, or does not reference a server handle, the *lppvData* pointer passed to this function will be initialized to a value of NULL and the function will return a value of zero.

If this function is called with a valid socket handle and there is no data associated with the socket, the function will return a non-zero value and the *lppvData* pointer will be returned with a NULL value. Before dereferencing the pointer returned by this function, the application should always check the return value to ensure the function succeeded and make sure that the pointer is not NULL.

Example

```
UINT *pnValue1 = (UINT *)LocalAlloc(LPTR, sizeof(UINT));  
UINT *pnValue2 = NULL;  
  
*pnValue1 = 1234;  
  
if (InetSetServerData(hServer, pnValue1) == FALSE)  
{  
    // Unable to associate the data with this server  
    return;  
}  
  
if (InetGetServerData(hServer, &pnValue2) == FALSE)  
{  
    // Unable to retrieve the data associated with this server  
    return;  
}
```

```
// *pnValue2 == 1234  
printf("The value of user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetClientData](#), [InetSetClientData](#), [InetSetServerData](#)

InetGetServerPriority Function

```
INT WINAPI InetGetServerPriority(  
    SOCKET hServer  
);
```

The **InetGetServerPriority** function returns the current priority for the specified server.

Parameters

hServer
Handle to the server socket.

Return Value

If the function succeeds, the return value is the priority for the specified server. If the function fails, the return value is INET_PRIORITY_INVALID. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerPriority** function can be used to determine the current priority assigned to the specified server. It will be one of the following values:

Value	Description
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

The socket handle for the server must be one that was created using the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSKV12.lib

See Also

[InetGetClientPriority](#), [InetServerStart](#), [InetSetClientPriority](#), [InetSetServerPriority](#)

InetGetServerStackSize Function

```
DWORD WINAPI InetGetServerStackSize(  
    SOCKET hServer  
);
```

Return the initial size of the stack allocated for threads created by the server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the amount of memory that will be allocated for the stack in bytes. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerStackSize** function returns the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSKV12.lib

See Also

[InetGetMemoryUsage](#), [InetServerStart](#), [InetSetServerStackSize](#)

InetGetServerStatus Function

```
INT WINAPI InetGetServerStatus(  
    SOCKET hServer  
);
```

The **InetGetServerStatus** function returns the current status of the specified server.

Parameters

hServer
Handle to the server socket.

Return Value

If the function succeeds, the return value is the status for the specified server. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetServerStatus** function can be used to determine the current status for the specified server. It may be one of the following values:

Value	Description
INET_SERVER_INACTIVE	The server is currently inactive.
INET_SERVER_STARTED	The server has initialized and is preparing to listen for client connections.
INET_SERVER_LISTENING	The server is actively listening for incoming client connections.
INET_SERVER_SUSPENDED	The server has been suspended and is no longer accepting client connections.
INET_SERVER_SHUTDOWN	The server has been stopped and is in the process of terminating all active client connections.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

- Minimum Desktop Platform:** Windows 7 Service Pack 1
- Minimum Server Platform:** Windows Server 2008 R2 Service Pack 1
- Header File:** cssock12.h
- Import Library:** cssock12.lib

See Also

[InetServerRestart](#), [InetServerResume](#), [InetServerStart](#), [InetServerStop](#), [InetServerSuspend](#)

InetGetServerThreadId Function

```
DWORD WINAPI InetGetServerThreadId(  
    SOCKET hServer  
);
```

The **InetGetServerThreadId** function returns the thread ID for the specified server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is an unsigned integer value which identifies the thread that was created to manage the server. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The thread ID returned by this function can be used with the **OpenThread** function to obtain a handle to the thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetClientPriority](#), [InetGetClientThreadId](#), [InetGetThreadClient](#), [InetSetClientPriority](#)

InetGetServiceName Function

```
BOOL WINAPI InetGetServiceName(  
    UINT nServicePort,  
    LPTSTR lpszServiceName,  
    INT nMaxLength  
);
```

The **InetGetServiceName** function returns the service name associated with a specified port number.

Parameters

nServicePort

Port number associated with some network service.

lpszServiceName

A null-terminated string buffer that will contain the name of the service associated with the specified port number. This string should be at least 32 characters long.

nMaxLength

An integer value which specifies the maximum number of characters that can be copied into the string buffer.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetServicePort](#)

InetGetServicePort Function

```
UINT WINAPI InetGetServicePort(  
    LPCTSTR lpszServiceName  
);
```

The **InetGetServicePort** function returns the port number associated with a service name.

Parameters

lpszServiceName

A null-terminated string which specifies the name of the service to return the port number for.

Return Value

If the function succeeds, the return value is the port number associated with a service name. If the function fails, the return value is INET_ERROR. To get extended error information, call

InetGetLastError.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetServiceName](#)

InetGetStatus Function

```
INT WINAPI InetGetStatus(  
    SOCKET hSocket  
);
```

The **InetGetStatus** function is used to report what sort of socket operation is in progress.

Parameters

hSocket
The socket handle.

Return Value

If the function succeeds, the return value is the client status code. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The return value is one of the following values:

Value	Description
INET_STATUS_UNUSED	No connection has been established.
INET_STATUS_IDLE	The socket is idle and not in a blocked state
INET_STATUS_LISTEN	The socket is listening for inbound connections from a client
INET_STATUS_CONNECT	The socket is establishing a connection with a server
INET_STATUS_ACCEPT	The socket is accepting a connection from a client
INET_STATUS_READ	Data is being read from the socket
INET_STATUS_WRITE	Data is being written to the socket
INET_STATUS_FLUSH	The socket is being flushed; all data in the receive buffers is being discarded
INET_STATUS_DISCONNECT	The socket is disconnecting from the remote host

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified socket.

Requirements

- Minimum Desktop Platform:** Windows 7 Service Pack 1
- Minimum Server Platform:** Windows Server 2008 R2 Service Pack 1
- Header File:** cswsock12.h
- Import Library:** cswskv12.lib

See Also

[InetGetBlockingSocket](#), [InetIsConnected](#), [InetIsListening](#), [InetIsReadable](#), [InetIsWritable](#)

InetGetStreamInfo Function

```
BOOL WINAPI InetGetStreamInfo(  
    SOCKET hSocket,  
    LPINETSTREAMINFO lpStreamInfo  
);
```

The **InetGetStreamInfo** function fills a structure with information about the current stream I/O operation.

Parameters

hSocket

Handle to the socket.

lpStreamInfo

A pointer to an [INETSTREAMINFO](#) structure which contains information about the status of the network stream.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetStreamInfo** function returns information about the current streaming socket operation, including the average number of bytes transferred per second and the estimated amount of time until the operation completes. If there is no operation currently in progress, this function will return the status of the last successful streaming read or write performed by the client.

In a multithreaded application, any thread in the current process may call this function to obtain status information for the specified socket.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetReadStream](#), [InetStoreStream](#), [InetWriteStream](#), [INETSTREAMINFO](#)

InetGetThreadClient Function

```
SOCKET WINAPI InetGetThreadClient(  
    DWORD dwThreadId  
);
```

The **InetGetThreadClient** function returns the socket handle for the client session that is being managed by the specified thread.

Parameters

dwThreadId

An unsigned integer value which identifies the thread managing the client session. If this parameter has a value of zero, then the client handle for the current thread is returned.

Return Value

If the function succeeds, the return value is the socket handle for the specified client session. If the function fails, the return value is `INVALID_SOCKET`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetThreadClient** is used to obtain the socket handle for the client session that is being managed by the specified thread. If the specified thread ID is zero, then the function will return the client socket for the current thread, otherwise it will search the internal table of all active client sessions and return the handle to the session that is being managed by that thread.

This function will fail if the thread ID does not specify an active client session thread.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetGetClientHandle](#), [InetGetClientId](#), [InetGetClientThreadId](#), [InetGetServerClient](#), [InetGetServerThreadId](#)

InetGetTimeout Function

```
INT WINAPI InetGetTimeout(  
    SOCKET hSocket  
);
```

The **InetGetTimeout** function returns the timeout interval for blocking operations, in seconds.

Parameters

hSocket

Handle to the socket.

Return Value

If the function succeeds, the return value is the timeout interval for blocking operations, in seconds. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

See Also

[InetSetTimeout](#)

InetGetUrlHostName Function

```
INT WINAPI InetGetUrlHostName(  
    LPCTSTR lpszUrl,  
    DWORD dwReserved,  
    LPTSTR lpszHostName,  
    INT nMaxLength,  
    UINT *lpnHostPort,  
    UINT *lpnProtocol,  
    DWORD *lpdwOptions  
);
```

The **InetGetUrlHostName** function returns the host name and port number specified in a URL.

Parameters

lpszUrl

A null-terminated string which specifies a URL. This parameter cannot be NULL or point to an empty string. If a non-standard URI scheme is used, the port number must be explicitly specified or the function will fail. See the remarks below for more information on the format supported by this function.

dwReserved

An unsigned integer value reserved for internal use. This value should always be zero.

lpszHostName

Pointer to the string buffer that will contain the canonical form of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer and must be large enough to store the complete host name.

nMaxLength

The maximum number of characters that can be copied to the *lpszHostName* string buffer. This parameter cannot be zero, and must include the terminating null character.

lpnHostPort

Pointer to an unsigned integer value which will contain the port number specified in the URL. This parameter value will always be initialized by the function with a value of zero. If this parameter is NULL, it will be ignored and no port information will be returned.

lpnProtocol

Pointer to an unsigned integer value which will contain the protocol associated with the URI scheme. This parameter value will always be initialized by the function with a value of zero. If this parameter is NULL, it will be ignored and no protocol information will be returned.

lpdwOptions

Pointer to an unsigned integer value which will contain any socket options required to establish a connection based on the URI scheme or specified port. This parameter value will always be initialized by the function with a value of zero. If this parameter is NULL, it will be ignored.

Return Value

If the function succeeds, the return value is the number of characters copied into the *lpszHostName* buffer. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetGetUrlHostName** function will extract the host name and port number from a URL and return the canonical form of the host name. If the *lpnHostPort*, *lpnProtocol* and *lpdwOptions* parameters have been specified, they will contain the port number, protocol and additional connection options associated with the URL scheme.

The general format of the URL should look like this:

```
[scheme]:// [[username : password] @] hostname [:port] / [path;parameters ...]
```

This function recognizes most standard URI schemes which use this format. The host name and port number specified in the URL will be used to establish a connection and the remaining information will be discarded. If the URL does not explicitly specify a port number, the default port number associated with the scheme will be used as the default value. For example, consider the following:

```
https://www.example.com/
```

In this example, there is no port number specified; instead, the default port for the **https://** scheme would be used, which is port 443. This function will also recognize a simpler format which only includes the host name and port number without a URI scheme, such as:

```
www.example.com:443
```

If the *lpzUrl* parameter only specifies a host name without a URI scheme or port number, this function will ignore the *lpnHostPort*, *lpnProtocol* and *lpdwOptions* parameters and return the canonical form of the host name in the *lpzHostName* string argument.

The host name portion of the URL can be specified as either a domain name or an IP address. Because an IPv6 address can contain colon characters, you must enclose the entire address in bracket [] characters. If this is not done, the function will return an error indicating the port number is invalid. For example, the URL **https://[2001:db8:0:0:1::128]/** uses an IPv6 host address and this would be considered valid. Without the brackets, this URL would not be accepted.

If the URL uses an IP address instead of a host name, this function will return a copy of that IP address in the *lpzHostName* string provided by the caller. The function will not attempt to resolve the IP address into a host name, however you can use the **InetGetHostName** function to perform a reverse DNS lookup if required.

The only URI schemes currently supported by this function use TCP stream connections. In practical terms, this means the *lpnProtocol* parameter will always return with the value `INET_PROTOCOL_TCP` when the function is successful. If the function fails, this value will be `INET_PROTOCOL_NONE`.

Although this function performs checks to ensure that the *lpzUrl* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the host name. To check if the host name exists and has a valid IP address, use the **InetValidateHostName** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetConnectUrl](#), [InetGetHostAddress](#), [InetGetHostName](#), [InetHostNameToUnicode](#), [InetNormalizeHostName](#)

InetGetVpnAdapter Function

```
BOOL WINAPI InetGetVpnAdapter(  
    LPTSTR lpszAdapterName,  
    INT nMaxLength,  
    INT nAddressFamily,  
    LPINTERNET_ADDRESS lpAddress  
);
```

The **InetGetVpnAdapter** function returns the name of the active VPN adapter.

Parameters

lpszAdapterName
A null-terminated string buffer which will contain the name of the active VPN adapter if one is found. If a VPN was not detected, the buffer will contain an empty string when the function returns. This parameter may be NULL if the adapter name is not required.

nMaxLength
The maximum number of characters which can be copied into the string buffer, including the terminating null character. If this value is zero, the ***lpszAdapterName*** parameter is ignored.

nAddressFamily
An integer value which identifies the type of IP address to return. It may be one of the following values:

Value	Description
INET_ADDRESS_ANY	Return the IP address for the adapter in either IPv4 or IPv6 format, depending on how the host name can be resolved. By default, a preference will be given for returning an IPv4 address. However, if the adapter only has an IPv6 address, that value will be returned.
INET_ADDRESS_IPV4	Specifies that the adapter address should be returned in IPv4 format. The first four bytes of the <i>ipNumber</i> member of the INTERNET_ADDRESS structure are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the adapter address should be returned in IPv6 format. All bytes in the <i>ipNumber</i> member of the INTERNET_ADDRESS structure are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

lpAddress
A pointer to an INTERNET_ADDRESS structure that will contain the IP address assigned to the VPN adapter when the function returns. This parameter may be NULL if this information is not required. If this parameter is NULL, the value of the ***nAddressFamily*** parameter is ignored.

Return Value

If the function succeeds, the return value is non-zero if an active VPN adapter was found. If no VPN adapter was detected, this function will return zero.

Remarks

The **InetGetVpnAdapter** function provides a best-effort method to detect if the system is currently using a Virtual Private Network (VPN) connection. The detection logic is based on identifying active network adapters with names or descriptions that match known VPN software or common VPN adapter naming conventions, while explicitly excluding adapters associated with virtualization platforms (such as Hyper-V, VMware, and VirtualBox), loopback interfaces, and other non-VPN virtual adapters.

Because there is no industry standard for how VPN clients are identified at the network adapter level, this function uses a heuristic approach and cannot guarantee detection in every scenario. Some VPN clients may use custom adapter names or attempt to obscure their presence, and new VPN software may not be recognized until the internal detection list is updated. There is also a small chance that non-VPN software using similar naming conventions could result in a false positive.

The result of this function should be treated as a hint for diagnostic or troubleshooting purposes. It should not be used to restrict access to your application or deny service to users based solely on the possible presence of a VPN connection. If you specify an address family and the *lpAddress* parameter is not NULL, it will contain the first IP address bound to the adapter interface. It is possible for VPN adapters to be bound to multiple IP addresses. The **InetFormatAddress** function can be used to convert the INTERNET_ADDRESS value to a printable string.

The adapter name returned by this function is the description of the detected VPN adapter, encoded as a UTF-8 string in the ANSI version of this function. If you require more detailed network information, consider using the **InetEnumNetworkAddresses** function. You can use the **InetGetExternalAddress** function to obtain the public-facing IP address allocated for the client.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetEnumNetworkAddresses](#), [InetFormatAddress](#), [InetGetAdapterAddress](#), [InetGetExternalAddress](#), [InetGetLocalAddress](#), [INTERNET_ADDRESS](#)

InetHostNameToUnicode Function

```
INT WINAPI InetHostNameToUnicode(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszUnicodeName,  
    INT nMaxLength  
);
```

The **InetHostNameToUnicode** function converts the canonical form of a host name to its Unicode version.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero-length string.

lpszUnicodeName

Pointer to the string buffer that will contain the original Unicode version of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer.

nMaxLength

The maximum number of characters that can be copied to the *lpszUnicodeName* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **DnsGetLastError**.

Remarks

The **InetHostNameToUnicode** function will convert the encoded ASCII version of a host name to its Unicode version. Although any valid host name is accepted by this function, it is intended to convert a Punycode encoded host name to its original Unicode character encoding.

If the Unicode version of this function is used, the value returned in *lpszUnicodeName* will be a Unicode string using UTF-16 encoding. If the ANSI version of this function, the value returned will be a Unicode string using UTF-8 encoding. To display a UTF-8 encoded host name, your application will need to convert it to UTF-16 using the **MultiByteToWideChar** function.

Although this function performs checks to ensure that the *lpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the [InetGetHostAddress](#) function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetHostAddress](#), [InetNormalizeHostName](#)

InetInitialize Function

```
BOOL WINAPI InetInitialize(  
    LPCTSTR lpszLicenseKey,  
    LPVOID lpvReserved  
);
```

The **InetInitialize** function initializes the library and validates the specified license key at runtime.

Parameters

lpszLicenseKey

Pointer to a string that specifies the runtime license key to be validated. If this parameter is NULL, the library will validate the development license installed on the local system.

lpvReserved

A reserved parameter. This value should always be NULL.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**. All other functions will fail until a license key has been successfully validated.

Remarks

This must be the first function that an application calls before using any of the other functions in the library. When a NULL license key is specified, the library will only function on the development system. Before redistributing the application to an end-user, you must ensure that this function is called with a valid license key.

The *lpvReserved* parameter is maintained for backward compatibility and should always be NULL. Applications should not pass a valid pointer or reference for this parameter. For languages that do not use C-style pointers, you can typically pass a value of zero for this parameter.

Although it is only required that **InetInitialize** be called once for the current process, it may be called multiple times; however, each call must be matched by a corresponding call to **InetUninitialize**.

This function dynamically loads other system libraries and allocates thread local storage. If you are calling the functions in this library from within another DLL, it is important that you do not call the **InetInitialize** or **InetUninitialize** functions from the **DllMain** function because it can result in deadlocks or access violation errors. If the DLL is linked with the C runtime library (CRT), it will automatically call the constructors and destructors for static and global C++ objects and has the same restrictions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetConnect](#), [InetConnectEx](#), [InetDisconnect](#), [InetListen](#), [InetUninitialize](#)

InetIsAddressNull Function

```
BOOL WINAPI InetIsAddressNull(  
    LPINTERNET_ADDRESS LpAddress  
);
```

The **InetIsAddressNull** function determines if the IP address is null.

Parameters

LpAddress

A pointer to an INTERNET_ADDRESS structure that contains the address to check.

Return Value

If the function succeeds and the IP address is null, or the *LpAddress* parameter is a NULL pointer, the return value is non-zero. If the function fails or the address is not null, the return value is zero. If the address family is not supported, the last error code will be updated. If the address is valid but not null, the last error code will be set to NO_ERROR.

Remarks

A null IP address is one where all bits for the address (32 bits for IPv4 or 128 bits for IPv6) are zero. This is a special address that is typically used when creating a passive socket that should listen for connections on all available network interfaces.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetAddress](#), [InetIsAddressRoutable](#), [INTERNET_ADDRESS](#)

InetIsAddressRoutable Function

```
BOOL WINAPI InetIsAddressRoutable(  
    LPINTERNET_ADDRESS LpAddress  
);
```

The **InetIsAddressRoutable** function determines if the IP address is routable over the Internet.

Parameters

LpAddress

A pointer to an `INTERNET_ADDRESS` structure that contains the address to check. This parameter cannot be NULL.

Return Value

If the function succeeds and the IP address is routable over the Internet, the return value is non-zero. If the function fails or the address is not routable, the return value is zero. If the parameter is NULL, or the address family is not supported, the last error code will be updated. If the address is valid but not routable, the last error code will be set to `NO_ERROR`.

Remarks

A routable IP address is one that can be reached by anyone over the public Internet. These are also commonly referred to as "public addresses" which are typically assigned to networks and individual hosts by an Internet service provider. There are also certain addresses that are not routable over the Internet, and used to address systems over a local network or private intranet. This function can be used to determine if a given IP address is public (routable) or private.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetGetAddress](#), [InetGetExternalAddress](#), [InetIsAddressNull](#), [INTERNET_ADDRESS](#)

InetIsBlocking Function

```
BOOL WINAPI InetIsBlocking(  
    SOCKET hSocket  
);
```

The **InetIsBlocking** function is used to determine if the socket is performing a blocking operation.

Parameters

hSocket

Socket handle.

Return Value

If the socket is currently performing a blocking operation, the function returns a non-zero value. If the socket is not performing a blocking operation, or the socket handle is invalid, the function returns zero.

Remarks

This function is typically used to determine if an open socket that is being used by another thread is currently blocked. A socket may block when waiting to receive data from a remote host or while data is actively being exchanged. Because there can only be one blocking socket operation per thread, this function can be used to determine if a function such as **InetRead** or **InetWrite** would fail because another thread is currently sending or receiving data on that socket. This socket handle that is passed to this function does not need to be owned by the current thread.

It is important to note that if this function returns a non-zero value, it does not guarantee that a subsequent read or write on the socket will succeed. The application should always check the return value from functions such as **InetRead** and **InetWrite** to ensure they were successful.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cssock12.lib

See Also

[InetIsConnected](#), [InetIsReadable](#), [InetIsWritable](#), [InetRead](#), [InetWrite](#)

InetIsClosed Function

```
BOOL WINAPI InetIsClosed(  
    SOCKET hSocket  
);
```

The **InetIsClosed** function is used to determine if the remote host has closed its socket.

Parameters

hSocket

Socket handle.

Return Value

If the remote host has closed its socket, the function returns a non-zero value. If the remote host has not closed its connection, or the socket handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: csoskv12.lib

See Also

[InetIsConnected](#), [InetIsListening](#), [InetIsReadable](#), [InetIsWritable](#)

InetIsConnected Function

```
BOOL WINAPI InetIsConnected(  
    SOCKET hSocket  
);
```

The **InetIsConnected** function is used to determine if the socket is currently connected to a remote host.

Parameters

hSocket

Socket handle.

Return Value

If the socket is connected to a remote host, the function returns a non-zero value. If the socket is not connected, or the socket handle is invalid, the function returns zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

See Also

[InetIsClosed](#), [InetIsListening](#), [InetIsReadable](#), [InetIsWritable](#)

InetIsListening Function

```
BOOL WINAPI InetIsListening(  
    SOCKET hSocket  
);
```

The **InetIsListening** function determines if the socket is listening for connection requests.

Parameters

hSocket

Socket handle.

Return Value

If the socket is being used to listen for connection requests, the function returns a non-zero value. If the socket is not listening or the socket handle is invalid, the function returns zero.

Remarks

The **InetIsListening** function determines if the socket is being used in a server application to actively listen for incoming connection requests from client applications. A listening socket can be created using either the **InetAsyncListen** or **InetListen** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetAsyncListen](#), [InetIsReadable](#), [InetIsWritable](#), [InetIsConnected](#), [InetListen](#)

InetIsProtocolAvailable Function

```
BOOL WINAPI InetIsProtocolAvailable(  
    INT nAddressFamily,  
    INT nProtocol  
);
```

The **InetIsProtocolAvailable** function determines if the operating system supports creating a socket for the specified address family and protocol.

Parameters

nAddressFamily

An integer which identifies the address family that should be checked. It should be one of the following values:

Value	Description
INET_ADDRESS_IPV4	Specifies that the function should determine if it can create an Internet Protocol version 4 (IPv4) socket. This requires that the system have an IPv4 TCP/IP stack bound to at least one network adapter on the local system. All Windows systems include support for IPv4 by default.
INET_ADDRESS_IPV6	Specifies that the function should determine if it can create an Internet Protocol version 6 (IPv6) socket. This requires that the system have an IPv6 TCP/IP stack bound to at least one network adapter on the local system. Windows XP and Windows Server 2003 includes support for IPv6, however it is not installed by default. Windows Vista and later versions include support for IPv6 and enable it by default.

nProtocol

An integer which identifies the protocol that should be checked. It should be one of the following values:

Value	Description
INET_PROTOCOL_TCP	Specifies the Transmission Control Protocol. This protocol provides a reliable, bi-directional byte stream. This requires that the system be capable of creating a stream socket using the specified address family.
INET_PROTOCOL_UDP	Specifies the User Datagram Protocol. This protocol is message oriented, sending data in discrete packets. This requires that the system be capable of creating a datagram socket using the specified address family.

Return Value

If the system is capable of creating a socket using the specified address family and protocol, this function will return a non-zero value. If the combination of address family and protocol is not supported, this function will return a value of zero.

Remarks

The **InetIsProtocolAvailable** function is used to determine if the operating system supports creating a particular type of socket. Typically it is used by an application to determine if the system has an IPv6 TCP/IP stack installed and configured. By default, all Windows systems will have an IPv4 stack installed if the system has a network adapter. However, not all systems may have an IPv6 stack installed, particularly older Windows XP and Windows Server 2003 systems. Note that if an IPv6 stack is not installed, the library will not recognize IPv6 addresses and cannot resolve host names that only have an IPv6 (AAAA) record, even if the address or host name is valid.

Example

```
if (!InetIsProtocolAvailable(INET_ADDRESS_IPV6, INET_PROTOCOL_TCP))
{
    AfxMessageBox(_T("This system does not support IPv6"), MB_ICONEXCLAMATION);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetAddress](#), [InetGetHostAddress](#), [InetGetHostName](#)

InetIsReadable Function

```
BOOL WINAPI InetIsReadable(  
    SOCKET hSocket,  
    DWORD dwTimeout,  
    LPDWORD lpdwAvail  
);
```

The **InetIsReadable** function is used to determine if data is available to be read from the socket.

Parameters

hSocket

Socket handle.

dwTimeout

Timeout value in milliseconds. If the socket cannot be read within this time period, the function will return a value of zero. A timeout value of zero specifies that the socket should be polled without blocking the current thread.

lpdwAvail

A pointer to an unsigned integer which will contain the number of bytes available to read.

Return Value

If the current thread can read data from the socket without blocking, the function returns a non-zero value. If the current thread cannot read any data without blocking, the function returns zero.

Remarks

On some platforms, the value returned in *lpdwAvail* will not exceed the size of the receive buffer (typically 64K bytes). Because of differences between TCP/IP stack implementations, it is not recommended that your application exclusively depend on this value to determine the exact number of bytes available. Instead, it should be used as a general indicator that there is data available to be read.

If the connection is secure, the value returned in *lpdwAvail* will reflect the number of bytes available in the encrypted data stream. The actual amount of data available to the application after it has been decrypted will vary.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetIsClosed](#), [InetIsWritable](#), [InetPeek](#), [InetRead](#), [InetReadLine](#), [InetReadStream](#)

InetIsUrgent Function

```
BOOL WINAPI InetIsUrgent(  
    SOCKET hSocket  
);
```

The **InetIsUrgent** function determines if there is any out-of-band (OOB) data available to be read.

Parameters

hSocket

Handle to the socket.

Return Value

If there is out-of-band data, the return value is non-zero. If there is no out-of-band data, or an error occurs the return value is zero. To determine if an error has occurred, use the **InetGetLastError** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: csksk12.lib

See Also

[InetSetOption \(INET_OPTION_INLINE\)](#)

InetIsWritable Function

```
BOOL WINAPI InetIsWritable(  
    SOCKET hSocket,  
    DWORD dwTimeout  
);
```

The **InetIsWritable** function is used to determine if data can be written to the socket.

Parameters

hSocket

Socket handle.

dwTimeout

Timeout value in milliseconds. If the socket cannot be written to within this time period, the function will return a value of zero. A timeout value of zero specifies that the socket should be polled without blocking the current thread.

Return Value

If the current thread can write data to the socket within the timeout period, the function returns a non-zero value. The function will return zero if the socket send buffer is full.

Remarks

The **InetIsWritable** function cannot be used to determine the amount of data that can be sent to the remote host without blocking the current thread. A non-zero return value only indicates that the send buffer is not full and can accept some data. In most cases, it is recommended that larger blocks of data be broken into smaller logical blocks rather than attempting to send it all of the data at once. For very large streams of data, it is recommended that you use the **InetWriteStream** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

See Also

[InetIsReadable](#), [InetWrite](#), [InetWriteLine](#), [InetWriteStream](#)

InetListen Function

```
SOCKET WINAPI InetListen(  
    LPCTSTR lpszLocalAddress,  
    UINT nLocalPort  
);
```

The **InetListen** function creates a passive socket used to listen for connections from a client application.

This function has been deprecated and is included for backwards compatibility. Use the **InetServerStart** function to create a server application.

Parameters

lpszLocalAddress

A null-terminated string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

Return Value

If the function succeeds, the return value is a socket handle. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

The socket option INET_OPTION_REUSEADDRESS is enabled by default when calling the **InetListen** function. This allows an application to re-use a local address and port number when creating the listening socket. If this behavior is not desired, use the **InetListenEx** function instead.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

After the listening socket has been created, the application should then call the **InetAccept** function to wait for a client to establish a connection. For servers that need to handle multiple simultaneous client connections, it is recommended that the asynchronous functions be used.

Example

```
SOCKET hServer = INVALID_SOCKET;  
LPCTSTR lpszAddress = _T("192.168.0.48");
```

```
// Accept connections from clients that connect to
// address 192.168.0.48 on port 7000

hServer = InetListen(lpszAddress, 7000);
if (hServer == INVALID_SOCKET)
{
    DWORD dwError;
    TCHAR szError[256];

    dwError = InetGetLastError();
    InetGetErrorString(dwError, szError, 256);

    MessageBox(NULL, szError, NULL, MB_OK|MB_TASKMODAL);
    return;
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetAccept](#), [InetAcceptEx](#), [InetInitialize](#), [InetListenEx](#), [InetServerStart](#)

InetListenEx Function

```
SOCKET WINAPI InetListenEx(  
    LPCTSTR lpszLocalAddress,  
    UINT nLocalPort,  
    UINT nBacklog,  
    DWORD dwOptions  
);
```

The **InetListenEx** function creates a passive socket, and specifies the maximum number of connection requests that will be queued.

This function has been deprecated and is included for backwards compatibility. Use the **InetServerStart** function to create a server application.

Parameters

lpszLocalAddress

A null-terminated string which specifies the local IP address that the socket should be bound to. If this parameter is NULL or points to an empty string, a client may establish a connection using any valid network interface configured on the local system. If an address is specified, then a client may only establish a connection with the system using that address.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero. Port numbers less than 1024 are considered reserved ports and may require that the process execute with administrative privileges and/or require changes to the default firewall rules to permit inbound connections.

nBacklog

The maximum length of the queue allocated for pending client connections. A value of zero specifies that the size of the queue should be set to a maximum reasonable value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are supported:

Value	Description
INET_OPTION_NONE	No option specified. If the address and port number are in use by another application or a closed socket which was listening on this port is still in the TIME_WAIT state, the function will fail.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option enables a server application to listen for connections using the specified address and port number even if they were in use recently. This is typically used to enable an application to close the listening socket and immediately reopen it without getting an error that the address is in use.
INET_OPTION_EXCLUSIVE	This option specifies the local address and port number is

	for the exclusive use by the current process, preventing another application from forcibly binding to the same address. If another process has already bound a socket to the address provided by the caller, this function will fail.
INET_OPTION_RESERVEDPORT	This option specifies the listening socket should be bound to an unused port number less than 1024, which is typically reserved for well-known system services. If this option is specified, the process may require administrative privileges and firewall rules that will permit a client to establish a connection with the service.
INET_OPTION_NOINHERIT	This option prevents the socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.

Return Value

If the function succeeds, the return value is a socket handle. If the function fails, the return value is `INVALID_SOCKET`. To get extended error information, call **InetGetLastError**.

Remarks

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

If the `INET_OPTION_REUSEADDRESS` option is not specified, an error may be returned if a listening socket was recently created for the same local address and port number. By default, once a listening socket is closed there is a period of time that all applications must wait before the address can be reused (this is called the `TIME_WAIT` state). The actual amount of time depends on the operating system and configuration parameters, but is typically two to four minutes. Specifying this option enables an application to immediately re-use a local address and port number that was previously in use. Note that this does not permit more than one server to bind to the same address.

If the `INET_OPTION_EXCLUSIVE` option is specified, the local address and port number cannot be used by another process until the listening socket is closed. This can prevent another application from forcibly binding to the same listening address as your server. This option can be useful in determining whether or not another process is already bound to the address you wish to use, but it may also prevent your server application from restarting immediately, regardless if the `INET_OPTION_REUSEADDRESS` option has also been specified.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

After the listening socket has been created, the application should then call the **InetAccept** or **InetAcceptEx** function to wait for a client to establish a connection. For servers that need to handle multiple simultaneous client connections, it is recommended that the asynchronous functions be used.

Example

```

SOCKET hServer = INVALID_SOCKET;
LPCTSTR lpszAddress = _T("192.168.0.48");

// Accept connections from clients that connect to
// address 192.168.0.48 on port 7000 with a standard
// backlog of 5 connections

hServer = InetListenEx(lpszAddress,
                      7000,
                      INET_BACKLOG,
                      INET_OPTION_REUSEADDRESS);

if (hServer == INVALID_SOCKET)
{
    DWORD dwError;
    TCHAR szError[256];

    dwError = InetGetLastError();
    InetGetErrorString(dwError, szError, 256);

    MessageBox(NULL, szError, NULL, MB_OK|MB_TASKMODAL);
    return;
}

```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetAccept](#), [InetAcceptex](#), [InetListen](#), [InetServerStart](#)

InetMatchHostName Function

```
BOOL WINAPI InetMatchHostName(  
    LPCTSTR lpszHostName,  
    LPCTSTR lpszHostMask  
    BOOL bResolve  
);
```

The **InetMatchHostName** function matches a host name against one or more strings that may contain wildcards.

Parameters

lpszHostName

A null-terminated string which specifies the host name or IP address to match.

lpszHostMask

A null-terminated string which specifies one or more values to match against the host name. The asterisk character can be used to match any number of characters in the host name, and the question mark can be used to match any single character. Multiple values may be specified by separating them with a semicolon.

bResolve

A boolean value which specifies if the host name or IP address should be resolved when matching the host against the mask string. If this parameter is non-zero, two checks against the host mask string will be performed; once for the host name specified and once for its IP address. If this parameter is zero, then the match is made only against the host name string provided.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetMatchHostName** function provides a convenient way for an application to determine if a given host name matches one or more mask strings which may contain wildcard characters. For example, the host name could be "www.microsoft.com" and the host mask string could be "*.microsoft.com". In this example, the function would return a non-zero value indicating the host name matched the mask. However, if the mask string was "*.net" then the function would return zero, indicating that there was no match. Multiple mask values can be combined by separating them with a semicolon; for example, the mask "*.com;*.org" would match any host name in either the .com or .org top-level domains.

If an internationalized domain name (IDN) is specified, it will be converted internally to an ASCII string using Punycode encoding. The host mask will be matched against this encoded version of the host name, not its Unicode version.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetAddress](#), [InetAddress.getHostAddress](#), [InetAddress.getHostName](#), [InetAddress.getLocalAddress](#), [InetAddress.getPeerAddress](#)

InetNormalizeHostName Function

```
INT WINAPI InetNormalizeHostName(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszNormalized,  
    INT nMaxLength  
);
```

The **InetNormalizeHostName** function returns the canonical form of a host name in the specified buffer.

Parameters

lpszHostName

Pointer to the host name as a null-terminated string. This parameter cannot be a NULL pointer or a zero-length string.

lpszNormalized

Pointer to the string buffer that will contain the canonical form of the host name, including the terminating null character. It is recommended that this buffer be at least 256 characters in size. This parameter cannot be a NULL pointer and must be large enough to store the complete host name.

nMaxLength

The maximum number of characters that can be copied to the *lpszNormalized* string buffer. This parameter cannot be zero, and must include the terminating null character.

Return Value

If the function succeeds, the return value is the number of characters copied into the string buffer. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetNormalizeHostName** function will remove all leading and trailing whitespace characters from the host name and fold all upper-case characters to lower-case. If an internationalized domain name (IDN) containing Unicode characters is passed to this function, it will be converted to an ASCII compatible format for domain names.

The *lpszHostName* parameter should only specify a host name or IP address. If you want to support the use of URLs to establish a connection, use the **InetGetUrlHostName** function which has extended support for extracting the host name and port number specified in a URL.

If the Unicode version of this function is used, the host name will be converted from UTF-16 to UTF-8 and then processed. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended you use the Unicode version.

Although this function performs checks to ensure that the *lpszHostName* parameter is in the correct format and does not contain any illegal characters or malformed encoding, it does not validate the existence of the domain name. To check if the host name exists and has a valid IP address, use the **InetValidateHostName** function.

It is recommended that you use this function if your application needs to store the host name, and if accepts a host name as user input. It is not necessary to call this function prior to calling other SocketWrench functions which accept a host name as a parameter. Those functions already normalize the host name and perform checks to ensure it is in the correct format.

If the *lpSzHostName* parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this function will return a copy of that IP address in the buffer provided by the caller. This allows the function to be used in cases where a user may input either a host name or IP address. To determine if the IP address has a corresponding host name, use the **InetGetHostName** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetHostAddress](#), [InetGetHostName](#), [InetGetUrlHostName](#), [InetHostNameToUnicode](#),
[InetValidateHostName](#)

InetPeek Function

```
INT WINAPI InetPeek(  
    SOCKET hSocket,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **InetPeek** function reads the specified number of bytes from the socket and copies them into the buffer, but it does not remove the data from the internal socket buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hSocket

The socket handle.

lpBuffer

Pointer to the buffer in which the data will be copied. This argument may be NULL, in which case no data is copied from the socket buffers, however the function will return the number of bytes available to read.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. If the *lpBuffer* parameter is not NULL, this value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes available to read from the socket. A return value of zero indicates that there is no data available to read at that time. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetPeek** function returns data that is available to read from the socket, up to the number of bytes specified. The data returned by this function is not removed from the socket buffers. It must be consumed by a subsequent call to the **InetRead** or **InetReadEx** function. The return value indicates the number of bytes that can be read in a single operation, up to the specified buffer size. However, it is important to note that it may not indicate the total amount of data available to be read from the socket at that time.

If no data is available to be read, the method will return a value of zero. Using this method in a loop to poll a non-blocking socket may cause the application to become non-responsive. To determine if there is data available to be read, use the **InetIsReadable** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cssock12.lib

See Also

[InetFlush](#), [InetRead](#), [InetReadEx](#), [InetWrite](#), [InetWriteEx](#)

InetRead Function

```
INT WINAPI InetRead(  
    SOCKET hSocket,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **InetRead** function reads the specified number of bytes from the socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hSocket

The socket handle.

lpBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the remote host has closed the connection and there is no more data available to be read. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetRead** function will read up to the specified number of bytes and store the data in the buffer provided by the caller. If there is no data available to be read at the time this function is called, the current thread will block until at least one byte of data becomes available, the timeout period elapses or an error occurs. This function will return if any amount of data is sent by the remote host, and will not block until the entire buffer has been filled. To avoid blocking the current thread, either create an asynchronous socket or use the **InetIsReadable** function to determine if there is data available to be read prior to calling this function.

The application should never make an assumption about the amount of data that will be available to read. TCP considers all data to be an arbitrary stream of bytes and does not impose any structure on the data itself. For example, if the remote host is sending data to the server in fixed 512 byte blocks of data, it is possible that a single call to the **Read** function will return only a partial block of data, or it may return multiple blocks combined together. It is the responsibility of the application to buffer and process this data appropriately.

For applications that are built using the Unicode character set, it is important to note that the buffer is an array of bytes, not characters. If the remote host is writing string data to the socket, it must be read as a stream of bytes and converted using the **MultiByteToWideChar** function. If the remote host is sending lines of text terminated with a linefeed or carriage return and linefeed pair, the **InetReadLine** function will return a line of text at a time and perform this conversion for you.

When **InetRead** is called and the socket is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is

available.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cswskv12.lib

See Also

[InetIsReadable](#), [InetPeek](#), [InetReadEx](#), [InetWrite](#), [InetWriteEx](#)

InetReadEx Function

```
INT WINAPI InetReadEx(  
    SOCKET hSocket,  
    LPVOID lpvBuffer,  
    INT cbBuffer,  
    DWORD dwReserved,  
    LPINTERNET_ADDRESS lpRemoteAddress,  
    UINT *lpnRemotePort  
);
```

The **InetReadEx** function reads the specified number of bytes from the socket and copies them into the buffer. The data may be of any type, and is not terminated with a null character.

Parameters

hSocket

The socket handle.

lpvBuffer

Pointer to the buffer in which the data will be copied.

cbBuffer

The maximum number of bytes to read and copy into the specified buffer. This value must be greater than zero.

dwReserved

Reserved parameter. This value must always be zero.

lpRemoteAddress

Pointer to an [INTERNET_ADDRESS](#) structure which will contain the IP address of the remote host that sent the data being read. If this information is not required, the parameter may be specified as NULL.

lpnRemotePort

Pointer to an unsigned integer which will contain the remote port number. If this information is not required, the parameter may be specified as NULL.

Return Value

If the function succeeds, the return value is the number of bytes actually read. A return value of zero indicates that the remote host has closed the connection and there is no more data available to be read. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

When **InetReadEx** is called and the socket is in non-blocking mode, it is possible that the function will fail because there is no available data to read at that time. This should not be considered a fatal error. Instead, the application should simply wait to receive the next asynchronous notification that data is available.

This function extends the **InetRead** function to return additional information about the peer who sent the data being received. For a client TCP socket, the IP address and remote port are the same values that were used to establish the connection. For a server TCP socket, it is the IP address and port number of the client which sent the data. When reading data from a UDP socket, this is the IP address

and remote port of the peer that sent the datagram. This information can be used in conjunction with the **InetWriteEx** function to send a datagram back to that host.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

See Also

[InetFlush](#), [InetGetPeerAddress](#), [InetPeek](#), [InetRead](#), [InetWrite](#), [InetWriteEx](#), [INTERNET_ADDRESS](#)

InetReadLine Function

```
BOOL WINAPI InetReadLine(  
    SOCKET hSocket,  
    LPTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

The **InetReadLine** function reads up to a line of data from the socket and returns it in a string buffer.

Parameters

hSocket

The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpszBuffer

Pointer to the string buffer that will contain the data when the function returns. The string will be terminated with a null byte, and will not contain the end-of-line characters.

lpnLength

A pointer to an integer value which specifies the length of the buffer. The value should be initialized to the maximum number of characters that can be copied into the string buffer, including the terminating null character. When the function returns, its value will be updated with the actual length of the string.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetReadLine** function reads data from the socket and copies into a specified string buffer. Unlike the **InetRead** function which reads arbitrary bytes of data, this function is specifically designed to return a single line of text data in a null-terminated string. When an end-of-line character sequence is encountered, the function will stop and return the data up to that point. The string buffer is guaranteed to be null-terminated and will not contain the end-of-line characters.

There are some limitations when using **InetReadLine**. The function should only be used to read text, never binary data. In particular, the function will discard nulls, linefeed and carriage return control characters. The Unicode version of this function will return a Unicode string, however this function does not support reading raw Unicode data from the socket. Any data read from the socket is internally buffered as octets (eight-bit bytes) and converted to Unicode using the **MultiByteToWideChar** function.

This function will force the thread to block until an end-of-line character sequence is processed, the read operation times out or the remote host closes its end of the socket connection. If this function is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **InetReadLine** is blocked waiting for data from the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **InetRead** and **InetReadLine** function calls can be intermixed, however be aware that **InetRead**

will consume any data that has already been buffered by the **InetReadLine** function and this may have unexpected results.

Unlike the **InetRead** function, it is possible for data to be returned in the buffer even if the return value is zero. Applications should also check the value of the *lpnLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the function is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the string buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the function return value.

Example

```
TCHAR szBuffer[MAXBUFLEN];
INT nLength;
BOOL bResult;

do
{
    nLength = sizeof(szBuffer);
    bResult = InetReadLine(hSocket, szBuffer, &nLength);

    if (nLength > 0)
    {
        // Process the line of data returned in the string
        // buffer; the string is always null-terminated
    }
} while (bResult);

DWORD dwError = InetGetLastError();
if (dwError == ST_ERROR_CONNECTION_CLOSED)
{
    // The remote host has closed its side of the connection and
    // there is no more data available to be read
}
else if (dwError != 0)
{
    // An error has occurred while reading a line of data
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetIsReadable](#), [InetRead](#), [InetWrite](#), [InetWriteLine](#)

InetReadStream Function

```
BOOL WINAPI InetReadStream(  
    SOCKET hSocket,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions,  
    LPBYTE lpMarker,  
    DWORD cbMarker,  
    DWORD dwReserved  
);
```

The **InetReadStream** function reads the socket data stream and stores the contents in the specified buffer.

Parameters

hSocket

The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpvBuffer

Pointer to the buffer that will contain or reference the data when the function returns. The actual argument depends on the value of the **dwOptions** parameter which specifies how the data stream will be stored.

lpdwLength

A pointer to an unsigned integer value which specifies the maximum length of the buffer and contains the number of bytes read when the function returns. This argument should always point to an initialized value. If the *lpvBuffer* argument specifies a memory buffer, then this argument cannot point to an initialized value of zero; if any other type of stream buffer is used and the initialized value is zero, that indicates that all available data from the socket should be returned until the end-of-stream marker is encountered or the remote host disconnects.

dwOptions

An unsigned integer value which specifies both the stream buffer type and any options to be used when reading the data stream. One of the following stream types may be specified:

Value	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the <i>lpvBuffer</i> parameter. If the argument specifies a pointer to a global memory handle initialized to NULL, then the function will return a handle which references the data; otherwise, the function will consider the parameter a pointer to a block of pre-allocated memory which will contain the stream data when the function returns. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The <i>lpvBuffer</i> argument specifies a pointer to a pre-allocated block of memory which will contain the data read from the socket when the function returns. If this stream buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned

	integer which has been initialized with the maximum length of the buffer.
INET_STREAM_HGLOBAL	The <i>lpvBuffer</i> argument specifies a pointer to a global memory handle. When the function returns, the handle will reference a block of memory that contains the stream data. The application should take care to make sure that the handle passed to the function does not currently reference a valid block of memory; it is recommended that the handle be initialized to NULL prior to calling this function.
INET_STREAM_HANDLE	The <i>lpvBuffer</i> argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read from the socket will be written to this handle using the WriteFile function.
INET_STREAM_SOCKET	The <i>lpvBuffer</i> argument specifies a socket handle. The data read from the socket specified by the <i>hSocket</i> argument will be written to this socket. The socket handle passed to this function must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either attach the socket using the InetAttachSocket function or use the INET_STREAM_HANDLE stream buffer type instead.
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data returned in the buffer to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.
INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being returned as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the source data stream; if the application is using a pre-allocated memory buffer, this must be considered before calling this function.

lpMarker

A pointer to an array of bytes which marks the end of the data stream. When this byte sequence is encountered by the function, it will stop reading and return to the caller. The buffer will contain all of the data read from the socket up to and including the end-of-stream marker. If this argument is NULL, then the function will continue to read from the socket until the maximum buffer size is reached, the remote host closes its socket or an error is encountered.

cbMarker

An unsigned integer value which specifies the length of the end-of-stream marker in bytes. If the *lpMarker* parameter is NULL, then this value must be zero.

dwReserved

A reserved parameter. This value must always be zero.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetReadStream** function enables an application to read an arbitrarily large stream of data and store it in memory, write it to a file or even another socket. Unlike the **InetRead** method, which will return immediately when any amount of data has been read, **InetReadStream** will only return when the buffer is full as specified by the *lpdwLength* parameter, the logical end-of-stream marker has been read, the socket closed by the remote host or when an error occurs.

This function will force the thread to block until the operation completes. If this function is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **InetReadStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for data to be returned in the buffer even if the function returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was copied into the buffer. For example, if a timeout occurs while the function is waiting for more data to arrive on the socket, it will return zero; however, data may have already been copied into the buffer prior to the error condition. It is the responsibility of the application to process that data, regardless of the function return value.

Because **InetReadStream** can potentially cause the application to block for long periods of time as the data stream is being read, the function will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **InetRegisterEvent** function, and can obtain information about the current operation by calling the **InetGetStreamInfo** function.

Example

```
HGLOBAL hgb1Buffer = NULL; // Return data in a global memory buffer
DWORD cbBuffer = 102400;   // Read up to 100K bytes
BOOL bResult;

bResult = InetReadStream(hSocket,
                        &hgb1Buffer,
                        &cbBuffer,
                        INET_STREAM_HGLOBAL | INET_STREAM_CONVERT,
                        NULL, 0, 0);

if (bResult && cbBuffer > 0)
{
    LPBYTE lpBuffer = (LPBYTE)GlobalLock(hgb1Buffer);

    // Use data in the stream buffer
```

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

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetStreamInfo](#), [InetRead](#), [InetReadLine](#), [InetStoreStream](#), [InetWrite](#), [InetWriteLine](#), [InetWriteStream](#)

InetRegisterEvent Function

```
INT WINAPI InetRegisterEvent(  
    SOCKET hSocket,  
    UINT nEventId,  
    INETEVENTPROC LpEventProc,  
    DWORD_PTR dwParam  
);
```

The **InetRegisterEvent** function registers a callback function for the specified event.

Parameters

hSocket
Socket handle.

nEventId
An unsigned integer which specifies which event should be registered with the specified callback function. This parameter is ignored if the socket handle specifies a server created using the **InetServerStart** function. One or more of the following values may be used:

Value	Description
INET_EVENT_ACCEPT	A network event that indicates the process has received a connection request from a client and should accept the connection using the InetAsyncAccept function. This event is only generated for server applications which have created an asynchronous socket using the InetAsyncListen function.
INET_EVENT_CONNECT	A network event that indicates the connection to the remote host has completed.
INET_EVENT_DISCONNECT	A network event that indicates the remote host has closed the connection. The process should read any remaining data and disconnect.
INET_EVENT_READ	A network event which indicates data is available to read. No additional messages will be posted until the process has read at least some of the data from the socket. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_WRITE	A network event which indicates the application can send data to the remote host. This notification is sent after a connection has been established, or after a previous attempt to write data has failed because it would result in a blocking operation. This event is only generated if the socket is in asynchronous mode.
INET_EVENT_TIMEOUT	The network operation has exceeded the specified timeout period. The application may attempt to retry the operation, or may disconnect from the remote host and report an error to the user.
INET_EVENT_CANCEL	The application has canceled a blocking operation. This event is fired once an operation has been terminated by the InetCancel function, and control has been returned to the

lpEventProc

Specifies the address of the application defined callback function. For more information about the callback function, see the description of the **InetEventProc** callback function. If this parameter is NULL, the callback for the specified event is disabled. This parameter cannot be NULL if the socket handle specifies a server created using the **InetServerStart** function.

dwParam

A user-defined integer value that is passed to the callback function. If the application targets the x86 (32-bit) platform, this parameter must be a 32-bit unsigned integer. If the application targets the x64 (64-bit) platform, this parameter must be a 64-bit unsigned integer.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is **INET_ERROR**. To get extended error information, call **InetGetLastError**.

Remarks

The **InetRegisterEvent** function associates a callback function with a specific event. The event handler is an **InetEventProc** function that is invoked when the event occurs. Arguments are passed to the function to identify the client session, the event type and the user-defined value specified when the event handler is registered. If the event occurs because of an error condition, the error code will be provided to the handler.

The callback function specified by the *lpEventProc* parameter must be declared using the **__stdcall** calling convention. This ensures the arguments passed to the event handler are pushed on to the stack in the correct order. Failure to use the correct calling convention will corrupt the stack and cause the application to terminate abnormally.

This function can be used to change the callback function and user defined parameter for a server created using the **InetServerStart** function. However, it cannot be used with client sockets automatically created by the server interface. Those sockets are managed separately in their own thread, and individual client event notifications are not supported.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

See Also

[InetAsyncAccept](#), [InetAsyncAcceptEx](#), [InetAsyncListen](#), [InetDisableEvents](#), [InetEnableEvents](#), [InetEventProc](#), [InetFreezeEvents](#)

InetReject Function

```
BOOL WINAPI InetReject(  
    SOCKET hSocket  
);
```

The **InetReject** function is used to reject a client connection request.

Parameters

hSocket

Handle to a listening socket.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetReject** function rejects a pending client connection and the remote host will see this as the connection being aborted. If there are no pending client connections at the time, this function will immediately return with an error indicating that the operation would cause the thread to block.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetAbort](#), [InetAccept](#), [InetListen](#)

InetServerAsyncNotify Function

```
BOOL WINAPI InetServerAsyncNotify(  
    SOCKET hServer,  
    HWND hWnd,  
    UINT uMsg  
);
```

Enable or disable asynchronous notification of changes in server status.

Parameters

- hServer**
The socket handle.
- hWnd**
A handle to the window whose window procedure will receive the notification message.
- uMsg**
The user-defined message that will be sent to the notification window.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerAsyncNotify** function is used by an application to enable or disable asynchronous notifications. The message window is typically the main UI window and these notifications are used signal to the application that it should update the user interface. If the **hWnd** parameter is not NULL, it must specify a valid window handle and the user-defined message must have a value of **WM_USER** or higher. The application cannot specify a notification message that is reserved by the operating system. The pseudo-handle **HWND_BROADCAST** cannot be specified as the notification window. If the **hWnd** parameter is NULL, notifications for the specified server will be disabled.

When asynchronous notifications are enabled for a server, the server will post the user-defined message to the window whenever there is a change in status or after a client has connected or disconnected from the server. The **wParam** message parameter will contain the notification message and the **lParam** message parameter will contain the handle to the server or the unique client ID. The following notification messages are defined:

Value	Description
INET_NOTIFY_STARTUP	This notification is sent when the server has started and is preparing to accept client connections. This notification is only sent once, and only if asynchronous notifications are enabled immediately after the InetServerStart function is called. This message will not be sent once the server has begun accepting client connections or when notification messages are disabled and then subsequently re-enabled at a later time. The lParam message parameter will specify the handle to the server.
INET_NOTIFY_LISTEN	This notification is sent when the server is listening for client connections. This notification message may be sent to the

	application multiple times over the lifetime of the server. If the server was suspended, this notification will be sent after the application calls the InetServerResume function to resume accepting client connections. The <i>lParam</i> message parameter will specify the handle to the server.
INET_NOTIFY_SUSPEND	This notification is sent when the server suspends accepting new connections because the application has called either the InetServerSuspend or InetServerSuspendEx function. This notification message may be sent to the application multiple times over the lifetime of the server. The <i>lParam</i> message parameter will specify the handle to the server.
INET_NOTIFY_RESTART	This notification is sent when the server is restarted using the InetServerRestart function. Note that the server socket handle provided by the <i>lParam</i> message parameter will specify the new socket handle of the restarted server instance, not the original socket handle. The <i>lParam</i> message parameter will specify the handle to the server.
INET_NOTIFY_CONNECT	This notification is sent when the server accepts a client connection and the thread that manages the client session has begun processing network events for that client. This message notification will not be sent if the client connection is rejected by the server. The <i>lParam</i> message parameter will specify the unique ID of the client that connected to the server.
INET_NOTIFY_DISCONNECT	This notification is sent when the client disconnects from the server and the client socket has been closed. This notification message may not occur for each client session that is forced to terminate as the result of the server being stopped using the InetServerStop function. The <i>lParam</i> message parameter will specify the unique ID of the client that disconnected from the server.
INET_NOTIFY_SHUTDOWN	This notification is sent when the server thread is in the process of terminating. At the time the application processes this notification message, the server handle in <i>lParam</i> will reference the defunct server and cannot be used with other server functions. The <i>lParam</i> message parameter will specify the handle to the server.

If asynchronous notifications are enabled, you should never use those notifications as a replacement for an event handler. When an event occurs, the callback function that handles the event is invoked in the context of the thread that manages the client session. The application should exchange data with the client within that event handler and not in response to a notification message. These notification messages should only be used to update the application UI in response to changes in the status of the server.

The INET_NOTIFY_CONNECT and INET_NOTIFY_DISCONNECT notifications are different from the other server notifications because the *lParam* message parameter does not specify the server handle, but rather the unique client ID associated with the session that connected to or disconnected from the server. If you need to obtain the handle to the client session using the ID, call the

InetGetClientHandle function. To obtain the server handle in response to the INET_NOTIFY_CONNECT message, use the **InetGetClientServerById** function. Note that at the time the application processes the INET_NOTIFY_DISCONNECT notification message, the client session will have already terminated.

This function can only be used with a handle returned by the **InetServerStart** function and cannot be used with sockets created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

See Also

[InetGetClientServerById](#), [InetGetServerStatus](#), [InetServerStart](#)

InetServerBroadcast Function

```
INT WINAPI InetServerBroadcast(  
    SOCKET hServer,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **InetServerBroadcast** function sends data to clients that are connected to the specified server.

Parameters

hServer

The socket handle.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the server clients.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of clients that the data was sent to. If the function fails, the return value is `INET_ERROR`. To get extended error information, call

InetGetLastError.

Remarks

The **InetServerBroadcast** function sends the contents of the buffer to all of the clients that are connected to the specified server. This function will block until all clients have been sent a copy of the data. There is no guarantee in which order the clients will receive and process the data that has been broadcast.

This function can only be used with a socket handle created using the **InetServerStart** function and cannot be used with sockets created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetClientBroadcast](#), [InetWrite](#), [InetWriteLine](#)

InetServerLock Function

```
BOOL WINAPI InetServerLock(  
    SOCKET hServer  
);
```

The **InetServerLock** function locks the specified server, causing other client threads to block until it is unlocked.

Parameters

hServer

The socket handle to the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerLock** function causes the specified server to enter a locked state where only the current thread may interact with the server and the clients that are connected to it. While a server is locked, all other threads will block when they attempt to perform a network operation. When the server is unlocked, the blocked threads will resume normal execution.

This function should be used carefully, and a server should never be left in a locked state for an extended period of time. It is meant to be used when the server process updates a global data structure and it must prevent any other threads from performing a network operation during the update. Only one server can be locked at any one time, and once a server has been locked, it can only be unlocked by the same thread.

The program should always check the return value from this function, and should never assume that the lock has been established. If more than one thread attempts to lock a server at the same time, there is no guarantee as to which thread will actually establish the lock. If a potential deadlock situation is detected, this function will fail and return a value of zero.

Every time the **InetServerLock** function is called, an internal lock counter is incremented, and the lock will not be released until the lock count drops to zero. This means that each call to **InetServerLock** must be matched by an equal number of calls to the **InetServerUnlock** function. Failure to do so will result in the server becoming non-responsive as it remains in a locked state.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetLockedServer](#), [InetServerUnlock](#)

InetServerRestart Function

```
SOCKET WINAPI InetServerRestart(  
    SOCKET hServer  
);
```

The **InetServerRestart** function restarts the server, terminating all active client sessions.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is the new socket handle for the specified server. If the function fails, the return value is `INVALID_SOCKET`. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerRestart** function will restart the specified server, terminating all active client sessions and recreating the listening socket. The socket handle that is returned by the function is the handle for the new listening socket, and the old handle value is no longer valid. If the function is unable to recreate the listening socket for any reason, the server thread is terminated.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

[InetGetServerStatus](#), [InetServerResume](#), [InetServerStart](#), [InetServerStop](#), [InetServerSuspend](#)

InetServerResume Function

```
BOOL WINAPI InetServerResume(  
    SOCKET hServer  
);
```

The **InetServerResume** function resumes accepting client connections on the specified server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerResume** function instructs the server to resume accepting client connections. Any pending client connections that were requested while the server was suspended will be accepted.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetServerStatus](#), [InetServerRestart](#), [InetServerStart](#), [InetServerStop](#), [InetServerSuspend](#)

InetServerStart Function

```
SOCKET WINAPI InetServerStart(  
    LPCTSTR lpszLocalHost,  
    UINT nLocalPort,  
    UINT nBacklog,  
    UINT nMaxClients,  
    UINT nTimeout,  
    UINT nPriority,  
    DWORD dwOptions,  
    INETEVENTPROC lpEventProc,  
    DWORD_PTR dwEventParam,  
    LPSECURITYCREDENTIALS lpCredentials  
);
```

The **InetServerStart** function begins listening for client connections on the specified local address and port number. The server is started in its own thread and manages the client sessions independently of the calling thread. All interaction with the server and its client sessions takes place inside the callback function specified by the caller.

Parameters

lpszLocalHost

A null-terminated string which specifies the local hostname or IP address address that the socket should be bound to. If this parameter is NULL or an empty string, then an appropriate address will automatically be used. A specific address should only be used if it is required by the application.

nLocalPort

The local port number that the socket should be bound to. This value must be greater than zero.

nBacklog

The maximum length of the queue allocated for pending client connections. A value of zero specifies that the size of the queue should be set to a maximum reasonable value. On Windows server platforms, the maximum value is large enough to queue several hundred pending connections.

nMaxClients

The maximum number of client connections that can be established with the server. A value of zero specifies that there should not be any fixed limit on the number of active client connections. This value can be adjusted after the server has been created by calling the **InetServerThrottle** function.

nTimeout

The number of seconds the server should wait for a client to perform a network operation. If the client does not exchange any information with the server within this period of time, a timeout event will occur. The timeout value affects all clients that are connected to the server.

nPriority

An integer value which specifies the priority for the server and all client sessions. The priority for a specific client session may be modified by calling the **InetSetClientPriority** function. This parameter may be one of the following values:

Value	Description
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. It is recommended that most

	applications use this priority.
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the client session. It is typically used with lightweight services running in the background that are designed for few client connections. The client thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the client session and meters the processor utilization for the client session. The client thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the client session and the thread will be given higher scheduling priority. It can be used when it is important for the client session thread to be highly responsive. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The thread will be given higher scheduling priority and will be more responsive to the remote host. It is not recommended that this priority be used on a system with a single processor.

dwOptions

An unsigned integer used to specify one or more socket options. The following values are supported:

Value	Description
INET_OPTION_NONE	No option specified. If the address and port number are in use by another application or a closed socket which was listening on this port is still in the TIME_WAIT state, the function will fail.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option enables a server application to listen for connections using the specified address and port number even if they were in use recently. This is typically used to enable an application to close the listening socket and immediately reopen it without getting an error that the address is in use.
INET_OPTION_KEEPALIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. Enabling this option will also help applications detect the physical loss of a network connection, such as an Ethernet cable being unplugged. This option does not guarantee that persistent

	connections will be maintained over long periods of time.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and combines smaller packets into a single larger packet when sending data to a remote host. Specifying this option can improve the responsiveness and overall throughput of applications that implement their own buffering and exchange large amounts of information.
INET_OPTION_NOINHERIT	This option prevents the server socket handle from being inherited by child processes created by the application. Using this option can mitigate situations in which a child process does not close the handle, leaving it open after the parent process has disconnected from the server.
INET_OPTION_SECURE	This option specifies that a secure connection should be established with the client, where the client immediately initiates the TLS handshake when it connects to the server. To implement an explicit TLS session, where the client establishes a standard, non-secure connection and then sends a command to the server to initiate a secure session, you should not use this option. Instead, use the InetEnableSecurity function to selectively enable TLS for the client session.

lpEventProc

Specifies the address of the application defined callback function. For more information about the callback function, see the description of the **InetEventProc** callback function. This parameter cannot be NULL.

dwEventParam

A user-defined integer value that is passed to the callback function.

lpCredentials

Pointer to credentials structure [SECURITYCREDENTIALS](#). This may be NULL, unless the *dwOptions* parameter includes INET_OPTION_SECURE. When a secure session is specified, the fields *dwSize*, *lpszCertStore*, and *lpszCertName* must be defined, while other fields may be left undefined. Set *dwSize* to the size of the **SECURITYCREDENTIALS** structure.

Return Value

If the function succeeds, the return value is a socket handle. If the function fails, the return value is INVALID_SOCKET. To get extended error information, call **InetGetLastError**.

Remarks

In most cases, the *lpszLocalHost* parameter should be a NULL pointer or an empty string. On a multihomed system, this will enable the server to accept connections on any appropriately configured network adapter. Specifying a hostname or IP address will limit client connections to that particular address. Note that the hostname or address must be one that is assigned to the local system, otherwise an error will occur.

If an IPv6 address is specified as the local address, the system must have an IPv6 stack installed and configured, otherwise the function will fail.

To listen for connections on any suitable IPv4 interface, specify the special dotted-quad address "0.0.0.0". You can accept connections from clients using either IPv4 or IPv6 on the same socket by specifying the special IPv6 address "::0", however this is only supported on Windows 7 and Windows Server 2008 R2 or later platforms. If no local address is specified, then the server will only listen for connections from clients using IPv4. This behavior is by design for backwards compatibility with systems that do not have an IPv6 TCP/IP stack installed.

If the `INET_OPTION_REUSEADDRESS` option is not specified, an error may be returned if a listening socket was recently created for the same local address and port number. By default, once a listening socket is closed there is a period of time that all applications must wait before the address can be reused (this is called the `TIME_WAIT` state). The actual amount of time depends on the operating system and configuration parameters, but is typically two to four minutes. Specifying this option enables an application to immediately re-use a local address and port number that was previously in use. Note that this does not permit more than one server to bind to the same address.

When the event handler callback function is invoked by the server, it normally executes in the context of the worker thread that manages that client session. This means that even if you do not explicitly create any threads in your application, you must design your program to be thread-safe, with synchronized access to global objects and data. If your application has a user interface, only the main UI thread should attempt to modify controls. If you attempt to modify a control from a worker thread, such as adding a row to a listbox control, it can result the application becoming deadlocked. This means that you should not attempt to directly update the UI from within the event handler function. To enable asynchronous server notifications for a GUI application, use the **`InetServerAsyncNotify`** function.

The socket handle returned by this function references the listening socket that was created when the server was started. The service is managed in another thread, and all interaction with the server and active client connections are performed inside the event handler. To disconnect all active connections, close the listening socket and terminate the server thread, call the **`InetServerStop`** function.

Example

```
#define SERVER_PORT      7000
#define SERVER_CLIENTS   100

SOCKET hServer = INVALID_SOCKET;

// Accept connections from clients that connection on port 7000 with a default
// backlog of 5 connections and a maximum of 100 client connections.

hServer = InetServerStart(NULL,
                          SERVER_PORT,
                          INET_BACKLOG,
                          SERVER_CLIENTS,
                          INET_TIMEOUT,
                          INET_PRIORITY_NORMAL,
                          INET_OPTION_REUSEADDRESS,
                          MyEventHandler,
                          0,
                          NULL);

if (hServer == INVALID_SOCKET)
{
    DWORD dwError;
    TCHAR szError[256];
```

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

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetServerData](#), [InetGetServerPriority](#), [InetGetServerStatus](#), [InetServerLock](#), [InetServerRestart](#),
[InetServerResume](#), [InetServerStop](#), [InetServerSuspend](#), [InetServerThrottle](#), [InetServerUnlock](#),
[InetSetServerData](#), [InetSetServerPriority](#) [InetValidateCertificate](#)

InetServerStop Function

```
BOOL WINAPI InetServerStop(  
    SOCKET hServer  
);
```

The **InetServerStop** function signals the server to stop listening for connections and terminates all client sessions.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerStop** function signals the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The socket handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

If this function is called when there is one or more clients connected to the server, it will signal each client thread to terminate and then wait for the server thread to terminate. As the client sessions are terminated, the event handler will not be invoked. If you wish to ensure that all clients are disconnected normally before stopping the server, call the **InetServerSuspendEx** function with the `INET_SUSPEND_DISCONNECT` option and then stop the server after the last client has disconnected.

Because the **InetServerStop** function waits for the server thread to terminate, this function may cause your application to block. If this is not desirable, use the **InetServerStopEx** function which can perform the shutdown sequence asynchronously.

After the server thread has been terminated, the listening socket will go into a TIME-WAIT state which prevents an application from reusing the same address and port number bound to that socket for a brief period of time, typically two to four minutes. This is normal behavior designed to prevent delayed or misrouted packets of data from being read by a subsequent connection. To immediately start a new server using the same local address and port number, the option `INET_OPTION_REUSEADDRESS` must be specified when calling the **InetServerStart** function.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: `cswsock12.h`

Import Library: `cswskv12.lib`

See Also

InetServerStopEx Function

```
BOOL WINAPI InetServerStopEx(  
    SOCKET hServer,  
    DWORD dwMilliseconds  
);
```

The **InetServerStopEx** function signals the server to stop listening for connections and terminates all client sessions.

Parameters

hServer

Handle to the server socket.

dwMilliseconds

An unsigned integer value that specifies the number of milliseconds to wait for all active clients to disconnect and the server thread to terminate.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerStopEx** function signals the server to stop accepting client connections, disconnects all active client connections and terminates the thread that is managing the server session. The socket handle is no longer valid after the server has been stopped and should no longer be used. Note that it is possible that the actual handle value may be re-used at a later point when a new server is started. An application should always consider the socket handle to be opaque and never depend on it being a specific value.

Unlike the **InetServerStop** function, which waits for fixed period of time for the server thread to terminate, the **InetServerStopEx** allows the caller to determine how much time should be spent waiting for the clients to disconnect and the server thread to terminate. If the *dwMilliseconds* parameter has a value of INFINITE, the function will wait for an indefinite period of time until all clients have disconnected, the listening socket closed and the server thread has terminated. If the *dwMilliseconds* parameter has a value of zero, the function does not wait for the server to shutdown. Instead, it returns immediately and the shutdown process continues in the background.

If your application specifies a value of zero for the *dwMilliseconds* parameter, the event handler will be invoked with the INET_EVENT_DISCONNECT event as each client disconnects from the server during the shutdown process. If you depend on this event to perform some cleanup on a per-client basis, you must ensure that the application does not exit until the server thread has terminated. To perform a graceful shutdown of the server, it is recommended that you use the **InetServerSuspendEx** function and specify the INET_SUSPEND_DISCONNECT option. After all clients have disconnected, call the **InetServerStop** function to terminate the server thread.

After the server thread has been terminated, the listening socket will go into a TIME-WAIT state which prevents an application from reusing the same address and port number bound to that socket for a brief period of time, typically two to four minutes. This is normal behavior designed to prevent delayed or misrouted packets of data from being read by a subsequent connection. To immediately start a new server using the same local address and port number, the option INET_OPTION_REUSEADDRESS must be specified when calling the **InetServerStart** function.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetServerStatus](#), [InetServerRestart](#), [InetServerStart](#), [InetServerStop](#), [InetServerSuspendEx](#)

InetServerSuspend Function

```
BOOL WINAPI InetServerSuspend(  
    SOCKET hServer  
);
```

Suspend accepting client connections on the specified server.

Parameters

hServer

Handle to the server socket.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerSuspend** function instructs the server to suspend accepting new client connections. Any incoming client connections will be queued up to the maximum backlog value specified when the server was started. To resume accepting client connections, call the **InetServerResume** function.

It is recommended that you only suspend a server if absolutely necessary, and only for brief periods of time. If you want to limit the number of active client connections or control the connection rate for clients, use the **InetServerThrottle** function.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSock12.lib

See Also

[InetGetServerStatus](#), [InetServerRestart](#), [InetServerResume](#), [InetServerStart](#), [InetServerStop](#), [InetServerThrottle](#)

InetServerThrottle Function

```
BOOL WINAPI InetServerThrottle(  
    SOCKET hServer,  
    UINT nMaxClients,  
    UINT nMaxClientsPerAddress,  
    DWORD dwConnectionRate  
);
```

The **InetServerThrottle** function limits the number of active client connections, connections per address and connection rate.

Parameters

hServer

Handle to the server socket.

nMaxClients

A value which specifies the maximum number of clients that may connect to the server. A value of zero specifies that there is no fixed limit to the number of client connections.

nMaxClientsPerAddress

A value which specifies the maximum number of clients that may connect to the server from the same IP address. A value of zero specifies that there is no fixed limit to the number of client connections per address. By default, there is no limit on the number of client connections per address.

dwConnectionRate

A value which specifies a restriction on the rate of client connections, limiting the number of connections that will be accepted within that period of time. A value of zero specifies that there is no restriction on the rate of client connections. The higher this value, the fewer the number of connections that will be accepted within a specific period of time. By default, there is no limit on the client connection rate.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerThrottle** function is used to limit the number of connections and the connection rate to minimize the potential impact of a large number of client connections over a short period of time. This can be used to protect the server from a client application that is malfunctioning or a deliberate denial-of-service attack in which the attacker attempts to flood the server with connection attempts.

If the maximum number of client connections or maximum number of connections per address is exceeded, the server will reject subsequent connection attempts until the number of active client sessions drops below the specified threshold. Note that adjusting these values lower than the current connection limits will not affect clients that have already connected to the server. For example, if the **InetServerStart** function is called with the maximum number of clients set to 100, and then **InetServerThrottle** is called lowering that value to 75, no existing client connections will be affected by the change. However, the server will not accept any new connections until the number of active clients drops below 75.

Increasing the connection rate value will force the server to slow down the rate at which it will accept

incoming client connection requests. For example, setting this parameter to a value of 1000 would limit the server to accepting one client connection every second, while a value of 250 would allow the server to accept four client connections per second. Note that significantly increasing the amount of time the server must wait to accept client connections can exceed the connection backlog queue, resulting in client connections being rejected.

The socket handle for the server must be one that was created by calling the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetServerStatus](#), [InetServerLock](#), [InetServerRestart](#), [InetServerResume](#), [InetServerStart](#), [InetServerSuspend](#), [InetServerUnlock](#)

InetServerUnlock Function

```
BOOL WINAPI InetServerUnlock(  
    SOCKET hServer  
);
```

The **InetServerUnlock** function unlock the specified server, allowing other client threads to resume execution.

Parameters

hServer

The socket handle to the server.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetServerUnlock** function releases the lock on the specified server and allows any blocked threads to resume execution. Only one server may be locked at any one time, and only the thread which established the lock can unlock the server.

Every time the **InetServerLock** function is called, an internal lock counter is incremented, and the lock will not be released until the lock count drops to zero. This means that each call to **InetServerLock** must be matched by an equal number of calls to the **InetServerUnlock** function. Failure to do so will result in the server becoming non-responsive as it remains in a locked state.

The program should always check the return value from this function, and should never assume that the lock has been released. If a potential deadlock situation is detected, this function will fail and return a value of zero.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cs wskv12.lib

See Also

[InetGetLockedServer](#), [InetGetServerStatus](#), [InetServerLock](#)

InetSetClientData Function

```
BOOL WINAPI InetSetClientData(  
    SOCKET hClient,  
    LPVOID lpvData  
);
```

The **InetSetClientData** function sets the application defined data associated with the specified client session.

Parameters

hClient

The client socket handle.

lpvData

Pointer to the application defined data associated with the specified client session.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the client session could not be modified. To get extended error information, call **InetGetLastError**.

Remarks

The **InetSetClientData** function is used to associate application defined data with a specific client session. This is typically used to associate a pointer to a data structure or a class instance with the client socket. The data can be retrieved using the **InetGetClientData** function.

You should never specify a pointer to a local variable or data structure that will go out of scope when the calling function exits. If you do this, the pointer will no longer be valid after the function exits and attempting to dereference that pointer at some later time can cause an exception to be thrown and terminate the program. You should always allocate a block of memory for the data using a function such as **HeapAlloc** or **LocalAlloc**. If you specify the address of a static or global data structure, you must use thread synchronization functions when dereferencing and modifying that structure.

This function can only be used with client socket handles created using the SocketWrench server interface. It cannot be used with socket handles created using the **InetConnect** or **InetAccept** functions.

Example

```
UINT *pnValue1 = (UINT *)LocalAlloc(LPTR, sizeof(UINT));  
UINT *pnValue2 = NULL;  
  
*pnValue1 = 1234;  
  
if (InetSetClientData(hClient, pnValue1) == FALSE)  
{  
    // Unable to associate the data with this session  
    return;  
}  
  
if (InetGetClientData(hClient, &pnValue2) == FALSE)  
{  
    // Unable to retrieve the data associated with this session  
    return;
```

```
}
```

```
// *pnValue2 == 1234  
printf("The value of user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetClientData](#), [InetGetServerData](#), [InetSetServerData](#)

InetSetClientMoniker Function

```
INT WINAPI InetSetClientMoniker(  
    SOCKET hSocket,  
    LPCTSTR lpszMoniker  
);
```

The **InetSetClientMoniker** function associates a unique string moniker with the specified client session.

Parameters

hSocket

Handle to the client socket.

lpszMoniker

Pointer to a string which specifies the moniker for the specified client socket. If this parameter is NULL or specifies an empty string, a moniker will no longer be associated with the client session.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

A client moniker is a string which can be used to uniquely identify a specific client session aside from its socket handle. The **InetGetClientMoniker** function will return the moniker that was previously assigned to the client, if any. To obtain the socket handle associated with a given moniker, use the **InetFindClientMoniker** function.

Monikers are not case-sensitive, and they must be unique so that no client socket for a particular server can have the same moniker. The maximum length for a moniker is 127 characters.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetFindClientMoniker](#), [InetGetClientHandle](#), [InetGetClientId](#), [InetGetClientMoniker](#)

InetSetClientPriority Function

```
INT WINAPI InetSetClientPriority(  
    SOCKET hClient,  
    INT nPriority  
);
```

The **InetSetClientPriority** function sets the current priority for the specified client session.

Parameters

hClient
Handle to the client session.

nPriority
An integer value which specifies the new priority for the client session. It may be one of the following values:

Value	Description
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. It is recommended that most applications use this priority.
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the client session. It is typically used with lightweight services running in the background that are designed for few client connections. The client thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the client session and meters the processor utilization for the client session. The client thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the client session and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The client thread will be given higher scheduling priority and will be more responsive to network events. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the function succeeds, the return value is the previous priority for the specified client session. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The **InetSetClientPriority** function can be used to change the current priority assigned to the specified client session. The client priority is inherited from the priority specified when the server is started using the **InetServerStart** function.

The socket handle for the client must be one that was created as part of the SocketWrench server interface, and cannot be a socket that was created using the **InetConnect** or **InetAccept** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetClientPriority](#), [InetGetServerPriority](#), [InetServerStart](#), [InetSetServerPriority](#)

InetSetHostFile Function

```
INT WINAPI InetSetHostFile(  
    LPCTSTR lpszFileName  
);
```

The **InetSetHostFile** function specifies the name of an alternate file to use when resolving hostnames and IP addresses. The host file is used as a database that maps an IP address to one or more hostnames, and is used by the **InetGetHostAddress** and **InetGetHostNames** function. The file is a plain text file, with each line in the file specifying a record, and each field separated by spaces or tabs. The format of the file must be as follows:

```
ipaddress hostname [hostalias ...]
```

For example, one typical entry maps the name "localhost" to the local loopback IP address. This would be entered as:

```
127.0.0.1 localhost
```

The hash character (#) may be used to specify a comment in the file, and all characters after it are ignored up to the end of the line. Blank lines are ignored, as are any lines which do not follow the required format.

Parameters

lpszFileName

Pointer to a string that specifies the name of the file. If the parameter is NULL, then the current host file is cleared from the cache and only the default host file will be used to resolve hostnames and addresses.

Return Value

If the function succeeds, the return value is the number of entries in the host file. A return value of INET_ERROR indicates failure. To get extended error information, call **InetGetLastError**.

Remarks

This function loads the file into memory allocated for the current thread. If the contents of the file have changed after the function has been called, those changes will not be reflected when resolving hostnames or addresses. To reload the host file from disk, call this function again with the same file name. To remove the alternate host file from memory, specify a NULL pointer as the parameter.

If a host file has been specified, it is processed before the default host file when resolving a hostname into an IP address, or an IP address into a hostname. If the host name or address is not found, or no host file has been specified, a nameserver lookup is performed.

To determine if an alternate host file has been specified, use the **InetGetHostFile** function. A return value of zero indicates that no alternate host file has been cached for the current thread.

A system may have a default host file, which is used to resolve hostnames before performing a nameserver lookup. To determine the name of this file, use the **InetGetDefaultHostFile** function. It is not necessary to specify this default host file, since it is always used to resolve host names and addresses.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetDefaultHostFile](#), [InetGetHostAddress](#), [InetGetHostFile](#), [InetGetHostName](#)

InetSetLastError Function

```
VOID WINAPI InetSetLastError(  
    DWORD dwErrorCode  
);
```

The **InetSetLastError** function sets the last-error code for the caller. This function is typically used to clear the last error by passing a value of zero as the parameter.

Parameters

dwErrorCode

Specifies the last-error code for the caller.

Return Value

None.

Remarks

Error codes are unsigned 32-bit values which are private to each calling thread. Most functions will set the last error code value when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value such as FALSE, NULL, INVALID_SOCKET or INET_ERROR. Those functions which call **InetSetLastError** when they succeed are noted on the function reference page.

Applications can retrieve the value saved by this function by using the [InetGetLastError](#) function. The use of **InetGetLastError** is optional; an application can call it to find out the specific reason for a function failure.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetErrorString](#), [InetGetLastError](#)

InetSetOption Function

```
INT WINAPI InetSetOption(  
    SOCKET hSocket,  
    DWORD dwOption,  
    BOOL bEnabled  
);
```

The **InetSetOption** function is used to enable or disable a specific socket option.

Parameters

hSocket

The socket handle.

dwOption

An unsigned integer used to specify one of the socket options. These options cannot be combined. The following values are recognized:

Value	Description
INET_OPTION_BROADCAST	This option specifies that broadcasting should be enabled for datagrams. This option is invalid for stream sockets.
INET_OPTION_KEEPAIVE	This option specifies that packets are to be sent to the remote system when no data is being exchanged to keep the connection active. This is only valid for stream sockets.
INET_OPTION_REUSEADDRESS	This option specifies the local address can be reused. This option is commonly used by server applications.
INET_OPTION_NODELAY	This option disables the Nagle algorithm, which buffers unacknowledged data and insures that a full-size packet can be sent to the remote host.

bEnabled

A boolean flag. If the flag is set to a non-zero value, the option is enabled. Otherwise the socket option is disabled.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

It is not recommend that you disable the Nagle algorithm by specifying the INET_OPTION_NODELAY flag unless it is absolutely required. Doing so can have a significant, negative impact on the performance of the application and network.

If if the INET_OPTION_KEEPAIVE option is enabled, keep-alive packets will start being generated five seconds after the socket has become idle with no data being sent or received. Enabling this option can be used by applications to detect when a physical network connection has been lost. However, it is recommended that most applications query the remote host directly to determine if the connection is still active. This is typically accomplished by sending specific commands to the server to query its status, or checking the elapsed time since the last response from the server.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetAsyncConnectEx](#), [InetConnectEx](#), [InetGetOption](#)

InetSetServerData Function

```
BOOL WINAPI InetSetServerData(  
    SOCKET hServer,  
    LPVOID LpvData  
);
```

The **InetSetServerData** function sets the application defined data associated with the specified server.

Parameters

hServer

The socket handle.

lpvData

Pointer to the application defined data associated with the specified server.

Return Value

If the function succeeds, the return value is non-zero. A return value of zero indicates that application defined data for the server could not be modified. To get extended error information, call **InetGetLastError**.

Remarks

The **InetSetServerData** function is used to associate application defined data with a specific server. The data can be retrieved using the **InetGetServerData** function.

You should never specify a pointer to a local variable or data structure that will go out of scope when the calling function exits. If you do this, the pointer will no longer be valid after the function exits and attempting to dereference that pointer at some later time can cause an exception to be thrown and terminate the program. You should always allocate a block of memory for the data using a function such as **HeapAlloc** or **LocalAlloc**. If you specify the address of a static or global data structure, you must use thread synchronization functions when dereferencing and modifying that structure.

This function can only be used with server socket handles created using the **InetServerStart** function. It cannot be used with socket handles created using the **InetListen** or **InetListenEx** functions.

Example

```
UINT *pnValue1 = (UINT *)LocalAlloc(LPTR, sizeof(UINT));  
UINT *pnValue2 = NULL;  
  
*pnValue1 = 1234;  
  
if (InetSetServerData(hServer, pnValue1) == FALSE)  
{  
    // Unable to associate the data with this server  
    return;  
}  
  
if (InetGetServerData(hServer, &pnValue2) == FALSE)  
{  
    // Unable to retrieve the data associated with this server  
    return;  
}  
  
// *pnValue2 == 1234
```

```
printf("The value of user defined data is %u\n", *pnValue2);
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: csoskv12.lib

See Also

[InetGetClientData](#), [InetGetServerData](#), [InetSetClientData](#)

InetSetServerPriority Function

```
INT WINAPI InetSetServerPriority(  
    SOCKET hServer,  
    INT nPriority  
);
```

The **InetSetServerPriority** function sets the current priority for the specified server.

Parameters

hServer

Handle to the server socket.

nPriority

An integer value which specifies the new priority for the server. It may be one of the following values:

Value	Description
INET_PRIORITY_BACKGROUND	This priority significantly reduces the memory, processor and network resource utilization for the server. It is typically used with lightweight services running in the background that are designed for few client connections. The server thread will be assigned a lower scheduling priority and will be frequently forced to yield execution to other threads.
INET_PRIORITY_LOW	This priority lowers the overall resource utilization for the server and meters the processor utilization for the server thread. The server thread will be assigned a lower scheduling priority and will occasionally be forced to yield execution to other threads.
INET_PRIORITY_NORMAL	The default priority which balances resource and processor utilization. This is the priority that is initially assigned to the server when it is started, and it is recommended that most applications use this priority.
INET_PRIORITY_HIGH	This priority increases the overall resource utilization for the server and the thread will be given higher scheduling priority. It is not recommended that this priority be used on a system with a single processor.
INET_PRIORITY_CRITICAL	This priority can significantly increase processor, memory and network utilization. The server thread will be given higher scheduling priority and will be more responsive to client connection requests. It is not recommended that this priority be used on a system with a single processor.

Return Value

If the function succeeds, the return value is the previous priority assigned to the server. If the function fails, the return value is INET_PRIORITY_INVALID. To get extended error information, call **InetGetLastError**.

Remarks

The **InetSetServerPriority** function can be used to change the current priority assigned to the specified server. Client connections that are accepted after this function is called will inherit the new priority as their default priority. Previously existing client connections will not be affected by this function. To modify the priority for an active client session, use the **InetSetClientPriority** function.

Higher priority values increase the thread priority and processor utilization for each client session. You should only change the server priority if you understand the impact it will have on the system and have thoroughly tested your application. Configuring the server to run with a higher priority can have a negative effect on the performance of other programs running on the system.

The socket handle for the server must be one that was created using the **InetServerStart** function, and cannot be a socket that was created using the **InetListen** or **InetListenEx** functions.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: csWSKV12.lib

See Also

[InetGetClientPriority](#), [InetGetServerPriority](#), [InetServerStart](#), [InetSetClientPriority](#)

InetSetServerStackSize Function

```
BOOL WINAPI InetSetServerStackSize(  
    SOCKET hServer,  
    DWORD dwStackSize  
);
```

Change the initial size of the stack allocated for threads created by the server.

Parameters

hServer

Handle to the server socket.

dwStackSize

The amount of memory that will be committed to the stack for each thread created by the server. If this value is zero, a default stack size will be used.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetSetServerStackSize** function changes the initial amount of memory that is committed to the stack for each thread created by the server. By default, the stack size for each thread is set to 256K for 32-bit processes and 512K for 64-bit processes. Increasing or decreasing the stack size will only affect new threads that are created by the server, it will not affect those threads that have already been created to manage active client sessions. It is recommended that most applications use the default stack size.

You should not change the stack size unless you understand the impact that it will have on your system and have thoroughly tested your application. Increasing the initial commit size of the stack will remove pages from the total system commit limit, and every page of memory that is reserved for stack cannot be used for any other purpose.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: csWSock12.h

Import Library: csWSKV12.lib

See Also

[InetGetMemoryUsage](#), [InetGetServerStackSize](#), [InetServerStart](#)

InetSetTimeout Function

```
INT WINAPI InetSetTimeout(  
    SOCKET hSocket,  
    UINT nTimeout  
);
```

The **InetSetTimeout** function sets the interval that is used when waiting for a blocking operation to complete.

Parameters

hSocket

Handle to the socket.

nTimeout

Duration of timeout interval, in seconds. If a value over 1000 is specified, it is assumed that milliseconds are intended by the user, and the value actually used will be adjusted accordingly.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetTimeout](#), [InetConnect](#), [InetAccept](#), [InetIsReadable](#), [InetIsWritable](#)

InetShutdown Function

```
INT WINAPI InetShutdown(  
    SOCKET hSocket,  
    DWORD dwOption  
);
```

The **InetShutdown** function is used to disable reception or transmission of data, or both.

Parameters

hSocket

The socket handle.

dwOption

An unsigned integer used to specify one of the shutdown options. These options cannot be combined. The following values are recognized:

Value	Description
INET_SHUTDOWN_READ	Disable reception of data.
INET_SHUTDOWN_WRITE	Disable transmission of data.
INET_SHUTDOWN_BOTH	Disable both reception and transmission of data.

Return Value

If the function succeeds, the return value is zero. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

This function is rarely needed. It is provided as an interface to the Windows Sockets **shutdown** function.

In some asynchronous applications, it may be desirable for a client to inform the server that no further communication is wanted, while allowing the client to read any residual data that may reside in internal buffers on the client side. **InetShutdown** accomplishes this because the socket handle is still valid after it has been called, although some or all communication with the remote host has ceased.

Requirements

- Minimum Desktop Platform:** Windows 7 Service Pack 1
- Minimum Server Platform:** Windows Server 2008 R2 Service Pack 1
- Header File:** cswsock12.h
- Import Library:** cswskv12.lib

See Also

[InetDisconnect](#)

InetStoreStream Function

```
BOOL WINAPI InetStoreStream(  
    SOCKET hSocket,  
    LPCTSTR lpszFileName,  
    DWORD dwLength,  
    LPDWORD lpdwCopied  
    DWORD dwOffset,  
    DWORD dwOptions  
);
```

The **InetStoreStream** function reads the socket data stream and stores the contents in the specified file.

Parameters

hSocket

The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpszFileName

Pointer to a string which specifies the name of the file to create or overwrite.

dwLength

An unsigned integer which specifies the maximum number of bytes to read from the socket and write to the file. If this value is zero, then the function will continue to read data from the socket until the remote host disconnects or an error occurs.

lpdwCopied

A pointer to an unsigned integer value which will contain the number of bytes written to the file when the function returns.

dwOffset

An unsigned integer which specifies the byte offset into the file where the function will start storing data read from the socket. Note that all data after this offset will be truncated. A value of zero specifies that the file should be completely overwritten if it already exists.

dwOptions

An unsigned integer value which specifies one or more options. Programs can use a bitwise operator to combine any of the following values:

Value	Description
INET_STREAM_CONVERT	The data stream is considered to be textual and will be modified so that end-of-line character sequences are converted to follow standard Windows conventions. This will ensure that all lines of text are terminated with a carriage-return and linefeed sequence. Because this option modifies the data stream, it should never be used with binary data. Using this option may result in the amount of data written to the file to be larger than the source data. For example, if the source data only terminates a line of text with a single linefeed, this option will have the effect of inserting a carriage-return character before each linefeed.

INET_STREAM_UNICODE	The data stream should be converted to Unicode. This option should only be used with text data, and will result in the stream data being written as 16-bit wide characters rather than 8-bit bytes. The amount of data returned will be twice the amount read from the source data stream. If the <i>dwOffset</i> parameter has a value of zero, the Unicode byte order mark (BOM) will be written to the beginning of the file.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetStoreStream** function enables an application to read an arbitrarily large stream of data and store it in a file. This function is essentially a simplified version of the **InetReadStream** function, designed specifically to be used with files rather than memory buffers or handles.

This function will force the thread to block until the operation completes. If this function is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, read the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **InetStoreStream** is blocked waiting for data from the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

Because **InetStoreStream** can potentially cause the application to block for long periods of time as the data stream is being read, the function will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **InetRegisterEvent** function, and can obtain information about the current operation by calling the **InetGetStreamInfo** function.

Example

```
DWORD dwCopied;
BOOL bResult;

bResult = InetStoreStream(hSocket,
                          lpzFileName,
                          &dwCopied,
                          0,
                          INET_STREAM_CONVERT);

if (bResult && dwCopied > 0)
{
    // The data has been written to the file
}
```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetGetStreamInfo](#), [InetRead](#), [InetReadLine](#), [InetReadStream](#), [InetWrite](#), [InetWriteLine](#),
[InetWriteStream](#)

InetUninitialize Function

```
VOID WINAPI InetUninitialize();
```

The **InetUninitialize** function terminates the use of the library.

Parameters

None.

Return Value

None.

Remarks

An application is required to perform a successful **InetInitialize** call before it can call any of the other library functions. When it has finished using the library, the application must call **InetUninitialize** to allow the library to free any resources allocated on behalf of the process. Any pending blocking or asynchronous calls in this process are canceled without posting any notification messages, and all sockets that were opened by the process are closed.

There should be a call to **InetUninitialize** for every successful call to **InetInitialize** made by a process. In a multithreaded environment, operations for all threads are terminated.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetDisconnect](#), [InetInitialize](#)

InetValidateCertificate Function

```
BOOL WINAPI InetValidateCertificate(  
    LPCTSTR lpszCertStore,  
    LPCTSTR lpszCertPassword,  
    LPCTSTR lpszCertName  
);
```

The **InetValidateCertificate** function determines if the specified security certificate is installed on the local system.

Parameters

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the personal certificate store will be used as the default. This parameter may also specify the name of a certificate file in PKCS #12 (PFX) format.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertPassword

A null-terminated string which specifies the password associated with a certificate file. This parameter is only used if the *lpszCertStore* parameter specifies a certificate file, otherwise it is ignored. If the certificate file is not protected with a password, this parameter should be a NULL pointer or empty string.

lpszCertName

A null-terminated string which specifies the name of the certificate to validate. The function will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the function will then attempt to find a certificate that has a matching common name (also called the certificate subject). This

comparison is less stringent, and the first partial match will be returned. If this second search fails, the function will return an error indicating that the certificate could not be found.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

If you are checking the validity of a certificate installed in the local certificate store, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is specified, then it will default to the certificate store for the current user.

It is possible to validate a certificate file rather than one stored in the local certificate store. The *lpzCertStore* member should specify the name of a file in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. If a password was specified when the certificate file was created, it must be provided in with the *lpzCertPassword* parameter or this function will be unable to access the certificate.

This function can only validate certificate files in PFX format and cannot be used to validate a certificate file in another format, such as PEM (base64 encoded) or DER (binary).

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetCreateSecurityCredentials](#), [InetDeleteSecurityCredentials](#)

InetValidateHostName Function

```
BOOL WINAPI InetValidateHostName(  
    LPCTSTR lpszHostName,  
    LPTSTR lpszAddress,  
    INT nMaxLength  
);
```

The **InetValidateHostName** function determines if the specified host name is valid and returns its IP address.

Parameters

lpszHostName

A null-terminated string which specifies the host name. The function will fail if this parameter is NULL or an empty string.

lpszAddress

A null-terminated string buffer which will contain the IP address of the host. If specified, this string must be large enough to store the complete IP address, including the terminating null character. If this parameter is NULL or the *nMaxLength* parameter is zero, it will be ignored and the IP address will not be returned.

nMaxLength

An integer value that specifies the maximum number of characters which can be copied into the *lpszAddress* string buffer. The buffer must be large enough to store the complete address. Because this function can return either an IPv4 or IPv6 address, it is recommended the minimum length for the buffer to be 46 characters. If this parameter is zero, the *lpszAddress* parameter will be ignored.

Return Value

If the function succeeds, the host name is valid and the return value will be non-zero. If the function fails, the host name could not be resolved to an IP address and the return value will be zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetValidateHostName** function provides a convenient way to determine if a host name is valid by attempting to resolve the name into an IP address. It is similar to calling the **InetNormalizeHostName** function to obtain the canonical form of the host name, calling **InetAddress** to obtain the IP address and then calling **InetFormatAddress** to return the string representation of the host's IP address.

If the Unicode version of this function is used, any non-ASCII characters in the host name will be automatically encoded into a compatible format and then resolved to an IP address. If you are unsure if an internationalized domain name will be specified as the host name, it is recommended you use the Unicode version.

The *lpszHostName* parameter can only specify a host name or IP address and cannot be a URL. If you want your application to support providing a URL in addition to a host name, use the **InetGetUrlHostName** function to extract the host name from the URL. You can then provide the host name to this function to obtain its IP address.

If the *lpszHostName* parameter specifies a valid IPv4 or IPv6 address string instead of a host name, this function will return a copy of that IP address in the buffer provided by the caller. This allows the function to be used in cases where a user may input either a host name or IP address. To determine if

the IP address has a corresponding host name, use the **InetGetHostName** function.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

Import Library: cssock12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetFormatAddress](#), [InetGetAddress](#), [InetGetHostAddress](#), [InetGetHostName](#), [InetGetUrlHostName](#),
[InetNormalizeHostName](#)

InetWrite Function

```
INT WINAPI InetWrite(  
    SOCKET hSocket,  
    LPBYTE lpBuffer,  
    INT cbBuffer  
);
```

The **InetWrite** function sends the specified number of bytes to the remote host.

Parameters

hSocket

The socket handle.

lpBuffer

The pointer to the buffer which contains the data that is to be sent to the remote host.

cbBuffer

The number of bytes to send from the specified buffer.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is INET_ERROR. To get extended error information, call **InetGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the application to send the remaining data at some later point. For non-blocking connections, the program must wait for the INET_EVENT_WRITE asynchronous notification message before it resumes sending data.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetFlush](#), [InetRead](#), [InetReadEx](#), [InetWriteEx](#)

InetWriteEx Function

```
INT WINAPI InetWriteEx(  
    SOCKET hSocket,  
    LPVOID lpvBuffer,  
    INT cbBuffer,  
    DWORD dwReserved,  
    LPINTERNET_ADDRESS lpRemoteAddress,  
    UINT nRemotePort  
);
```

The **InetWriteEx** function sends the specified number of bytes to the remote host.

Parameters

hSocket

The socket handle.

lpvBuffer

The pointer to the buffer which contains the data that is to be sent to the remote host.

cbBuffer

The number of bytes to send from the specified buffer.

dwReserved

Reserved parameter. This value must always be zero.

lpRemoteAddress

Pointer to an [INTERNET_ADDRESS](#) structure that specifies the address of the remote host that is to receive the data being written. For TCP stream sockets, this parameter must always be NULL or specify the same address that was used to establish the connection. For UDP datagram sockets, this may specify any valid IP address.

nRemotePort

The port number of the remote host that is to receive the data being written. For TCP stream sockets, this value must always be zero, or specify the same port number that was used to establish the connection. For UDP datagram sockets, this may specify any valid port number.

Return Value

If the function succeeds, the return value is the number of bytes actually written. If the function fails, the return value is `INET_ERROR`. To get extended error information, call **InetGetLastError**.

Remarks

The return value may be less than the number of bytes specified by the *cbBuffer* parameter. In this case, the data has been partially written and it is the responsibility of the application to send the remaining data at some later point. For non-blocking connections, the program must wait for the `INET_EVENT_WRITE` asynchronous notification message before it resumes sending data.

This function extends the **InetWrite** function to additional information about the destination IP address and port number for the data being written. For a client TCP connection, the IP address and remote port must be the same values that were used to establish the connection. When writing on a UDP socket, this is the IP address and remote port of the peer that will receive the datagram. This information can be used in conjunction with the **InetReadEx** function to send a datagram back to that host.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetFlush](#), [InetRead](#), [InetReadEx](#), [InetWrite](#), [INTERNET_ADDRESS](#)

InetWriteLine Function

```
BOOL WINAPI InetWriteLine(  
    SOCKET hSocket,  
    LPCTSTR lpszBuffer,  
    LPINT lpnLength  
);
```

The **InetWriteLine** function sends a line of text to the remote host, terminated by a carriage-return and linefeed.

Parameters

hSocket

The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpszBuffer

The pointer to a string buffer which contains the data that will be sent to the remote host. All characters up to, but not including, the terminating null character will be written to the socket. The data will always be terminated with a carriage-return and linefeed control character sequence. If this parameter points to an empty string or NULL pointer, then only a carriage-return and linefeed are written to the socket.

lpnLength

A pointer to an integer value which will contain the number of characters written to the socket, including the carriage-return and linefeed sequence. If this information is not required, a NULL pointer may be specified.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetWriteLine** function writes a line of text to the remote host and terminates the line with a carriage-return and linefeed control character sequence. Unlike the **InetWrite** function which writes arbitrary bytes of data to the socket, this function is specifically designed to write a single line of text data from a string.

If the *lpszBuffer* string is terminated with a linefeed (LF) or carriage return (CR) character, it will be automatically converted to a standard CRLF end-of-line sequence. Because the string will be sent with a terminating CRLF sequence, the value returned in the *lpnLength* parameter will typically be larger than the original string length (reflecting the additional CR and LF characters), unless the string was already terminated with CRLF.

There are some limitations when using **InetWriteLine**. The function should only be used to send text, never binary data. In particular, the function will discard nulls and append linefeed and carriage return control characters to the data stream. The Unicode version of this function will accept a Unicode string, however this function does not support writing raw Unicode data to the socket. Unicode strings will be automatically converted to UTF-8 encoding using the **WideCharToMultiByte** function and then written to the socket as a stream of bytes.

This function will force the thread to block until the complete line of text has been written, the write operation times out or the remote host aborts the connection. If this function is called with

asynchronous events enabled, it will automatically switch the socket into a blocking mode, send the data and then restore the socket to asynchronous operation. If another socket operation is attempted while **InetWriteLine** is blocked sending data to the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

The **InetWrite** and **InetWriteLine** function calls can be safely intermixed.

Unlike the **InetWrite** function, it is possible for data to have been written to the socket if the return value is zero. For example, if a timeout occurs while the function is waiting to send more data to the remote host, it will return zero; however, some data may have already been written prior to the error condition. If this is the case, the *lpnLength* argument will specify the number of characters actually written up to that point.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

Unicode: Implemented as Unicode and ANSI versions

See Also

[InetIsWritable](#), [InetRead](#), [InetReadLine](#), [InetWrite](#)

InetWriteStream Function

```
BOOL WINAPI InetWriteStream(  
    SOCKET hSocket,  
    LPVOID lpvBuffer,  
    LPDWORD lpdwLength,  
    DWORD dwOptions  
);
```

The **InetWriteStream** function writes data from the stream buffer to the specified socket.

Parameters

hSocket
The socket handle. The socket must reference a stream socket, not a datagram or raw socket. If the socket type is not valid, the function will return an error.

lpvBuffer
Pointer to the buffer that contains or references the data to be written to the socket. The actual argument depends on the value of the **dwOptions** parameter which specifies how the data stream will be accessed.

lpdwLength
A pointer to an unsigned integer value which specifies the size of the buffer and contains the number of bytes written when the function returns. This argument should should always point to an initialized value. If the **lpvBuffer** argument specifies a memory buffer or global memory handle, then this argument cannot point to an initialized value of zero.

dwOptions
An unsigned integer value which specifies the stream buffer type to be used when writing the data stream to the socket. One of the following stream types may be specified:

Value	Description
INET_STREAM_DEFAULT	The default stream buffer type is determined by the value passed as the lpvBuffer parameter. If the argument specifies a a global memory handle, then the function will write the data referenced by that handle; otherwise, the function will consider the parameter a pointer to a block of memory which contains data to be written. In most cases, it is recommended that an application explicitly specify the stream buffer type rather than using the default value.
INET_STREAM_MEMORY	The lpvBuffer argument specifies a pointer to a block of memory which contains the data to be written to the socket. If this stream buffer type is used, the lpdwLength argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HGLOBAL	The lpvBuffer argument specifies a global memory handle that references the data to be written to the socket. The handle must have been created by a call to the GlobalAlloc or GlobalReAlloc function. If this stream

	buffer type is used, the <i>lpdwLength</i> argument must point to an unsigned integer which has been initialized with the size of the buffer.
INET_STREAM_HANDLE	The <i>lpvBuffer</i> argument specifies a Windows handle to an open file, console or pipe. This should be the same handle value returned by the CreateFile function in the Windows API. The data read using the ReadFile function with this handle will be written to the socket.
INET_STREAM_SOCKET	The <i>lpvBuffer</i> argument specifies a socket handle. The data read from the socket specified by this handle will be written to the socket specified by the <i>hSocket</i> parameter. The socket handle passed to this function must have been created by this library; if it is a socket created by an third-party library or directly by the Windows Sockets API, you should either attach the socket using the InetAttachSocket function or use the INET_STREAM_HANDLE stream buffer type instead.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero. To get extended error information, call **InetGetLastError**.

Remarks

The **InetWriteStream** function enables an application to write an arbitrarily large stream of data from memory or a file to the specified socket. Unlike the **InetWrite** function, which may not write all of the data in a single function call, **InetWriteStream** will only return when all of the data has been written or an error occurs.

This function will force the thread to block until the operation completes. If this function is called with asynchronous events enabled, it will automatically switch the socket into a blocking mode, write the data stream and then restore the socket to asynchronous operation when it has finished. If another socket operation is attempted while **InetWriteStream** is blocked sending data to the remote host, an error will occur. It is recommended that this function only be used with blocking (synchronous) socket connections; if the application needs to establish multiple simultaneous connections, it should create worker threads to manage each connection.

It is possible for some data to have been written even if the function returns a value of zero. Applications should also check the value of the *lpdwLength* argument to determine if any data was sent. For example, if a timeout occurs while the function is waiting to write more data, it will return zero; however, some data may have already been written to the socket prior to the error condition.

Because **InetWriteStream** can potentially cause the application to block for long periods of time as the data stream is being written, the function will periodically generate INET_EVENT_PROGRESS events. An application can register an event handler using the **InetRegisterEvent** function, and can obtain information about the current operation by calling the **InetGetStreamInfo** function.

Example

```
HANDLE hFile;
```

```

DWORD dwLength;

hFile = CreateFile(lpszFileName,
                  GENERIC_READ,
                  FILE_SHARE_READ,
                  NULL,
                  OPEN_EXISTING,
                  FILE_FLAG_SEQUENTIAL_SCAN,
                  NULL);

if (hFile == INVALID_HANDLE_VALUE)
    return;

dwLength = GetFileSize(hFile, NULL);

if (dwLength > 0)
{
    BOOL bResult = InetWriteStream(
        hSocket,
        hFile,
        &dwLength,
        INET_STREAM_HANDLE);
}

CloseHandle(hFile);

```

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cswsock12.h

Import Library: cswskv12.lib

See Also

[InetGetStreamInfo](#), [InetRead](#), [InetReadLine](#), [InetReadStream](#), [InetStoreStream](#), [InetWrite](#), [InetWriteLine](#)

SocketWrench Data Structures

- INETSTREAMINFO
- INTERNET_ADDRESS
- SECURITYCREDENTIALS
- SECURITYINFO

Copyright © 2026 Catalyst Development Corporation. All rights reserved.

INETSTREAMINFO

The INETSTREAMINFO structure contains information about the data stream being currently read or written.

```
typedef struct _INETSTREAMINFO
{
    DWORD    dwStreamThread;
    DWORD    dwStreamSize;
    DWORD    dwStreamCopied;
    DWORD    dwStreamMode;
    DWORD    dwStreamError;
    DWORD    dwBytesPerSecond;
    DWORD    dwTimeElapsed;
    DWORD    dwTimeEstimated;
} INETSTREAMINFO, *LPINETSTREAMINFO;
```

Members

dwStreamThread

Specifies the numeric ID for the thread that created the socket.

dwStreamSize

The maximum number of bytes that will be read or written. This is the same value as the buffer length specified by the caller, and may be zero which indicates that no maximum size was specified. Note that if this value is zero, the application will be unable to calculate a completion percentage or estimate the amount of time for the operation to complete.

dwStreamCopied

The total number of bytes that have been copied to or from the stream buffer.

dwStreamMode

A numeric value which specifies the stream operation that is current being performed. It may be one of the following values:

Value	Description
INET_STREAM_READ	Data is being read from the socket and stored in the specified stream buffer.
INET_STREAM_WRITE	Data is being written from the specified stream buffer to the socket.

dwStreamError

The last error that occurred when reading or writing the data stream. If no error has occurred, this value will be zero.

dwBytesPerSecond

The average number of bytes that have been copied per second.

dwTimeElapsed

The number of seconds that have elapsed since the file transfer started.

dwTimeEstimated

The estimated number of seconds until the operation is completed. This is based on the average number of bytes transferred per second and requires that a maximum stream buffer size be specified by the caller.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: cssock12.h

See Also

[InetReadStream](#), [InetStoreStream](#), [InetWriteStream](#)

INTERNET_ADDRESS

This structure represents a numeric IPv4 or IPv6 address in network byte order.

```
typedef struct _INTERNET_ADDRESS
{
    INT     ipFamily;
    BYTE    ipNumber[16];
} INTERNET_ADDRESS, *LPINTERNET_ADDRESS;
```

Members

ipFamily

An integer which identifies the type of IP address. It will be one of the following values:

Value	Description
INET_ADDRESS_UNKNOWN	The address has not been specified or the bytes in the <i>ipNumber</i> array does not represent a valid address. Functions which populate this structure will use this value to indicate that the address cannot be determined.
INET_ADDRESS_IPV4	Specifies that the address is in IPv4 format. The first four bytes of the <i>ipNumber</i> array are significant and contains the IP address. The remaining bytes are not significant and an application should not depend on them having any particular value, including zero.
INET_ADDRESS_IPV6	Specifies that the address is in IPv6 format. All bytes in the <i>ipNumber</i> array are significant. Note that it is possible for an IPv6 address to actually represent an IPv4 address. This is indicated by the first 10 bytes of the address being zero.

ipNumber

A byte array which contains the numeric form of the IP address. This array is large enough to store both IPv4 (32 bit) and IPv6 (128 bit) addresses. The values are stored in network byte order.

Remarks

The INTERNET_ADDRESS structure is used by some functions to represent an Internet address in a binary format that is compatible with both IPv4 and IPv6 addresses. Applications that use this structure should consider it to be opaque, and should not modify the contents of the structure directly.

For compatibility with legacy applications that expect an IP address to be 32 bits and stored in an unsigned integer, you can copy the first four bytes of the *ipNumber* array using the **CopyMemory** function or equivalent. Note that if this is done, your application should always check the *ipFamily* member first to make sure that it is actually an IPv4 address.

Requirements

- Minimum Desktop Platform:** Windows 7 Service Pack 1
- Minimum Server Platform:** Windows Server 2008 R2 Service Pack 1
- Header File:** cswsock12.h

SECURITYCREDENTIALS

The **SECURITYCREDENTIALS** structure specifies the information needed by the library to specify additional security credentials, such as a client certificate, when establishing a secure connection.

```
typedef struct _SECURITYCREDENTIALS
{
    DWORD    dwSize;
    DWORD    dwProtocol;
    DWORD    dwOptions;
    DWORD    dwReserved;
    LPCTSTR  lpszHostName;
    LPCTSTR  lpszUserName;
    LPCTSTR  lpszPassword;
    LPCTSTR  lpszCertStore;
    LPCTSTR  lpszCertName;
    LPCTSTR  lpszKeyFile;
} SECURITYCREDENTIALS, *LPSECURITYCREDENTIALS;
```

Members

dwSize

Size of this structure. If the structure is being allocated dynamically, this member must be set to the size of the structure and all other unused structure members must be initialized to a value of zero or NULL.

dwProtocol

A bitmask of supported security protocols. The value of this structure member is constructed by using a bitwise operator with any of the following values:

Value	Description
SECURITY_PROTOCOL_DEFAULT	The default security protocol will be used when establishing a connection. This option will always request the latest version of the preferred security protocol and is the recommended value. Currently the default security protocol is TLS 1.2.
SECURITY_PROTOCOL_TLS	Either the TLS 1.2 or 1.3 protocol should be used when establishing a secure connection. The correct protocol is automatically selected, based on what version of the protocol is supported by the server. If this is the only protocol specified, SSL will be excluded from the list of supported protocols. This may be necessary for some servers which reject any attempt to use the older SSL protocol and require that only TLS be used.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol should be used when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol should be used when establishing

	a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol should be used when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol should be used when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwOptions

A value which specifies one or options. This member is constructed by using a bitwise operator with any of the following values:

Value	Description
CREDENTIAL_STORE_CURRENT_USER	The library will attempt to open a store for the current user using the certificate store name provided in this structure. If no options are specified, then this is the default behavior.
CREDENTIAL_STORE_LOCAL_MACHINE	The library will attempt to open a local machine store using the certificate store name provided in this structure. If this option is specified, the library will always search for a local machine store before searching for the store by that name for the current user.
CREDENTIAL_STORE_FILENAME	The certificate store name is a file that contains the certificate and private key that will be used.

dwReserved

This structure member is reserved for future use and should always be initialized to zero.

lpszHostName

A null-terminated string which specifies the hostname that will be used when validating the server certificate. If this member is NULL, then the server certificate will be validated against the hostname used to establish the connection.

lpszUserName

A null-terminated string which identifies the owner of client certificate. Currently this member is not used by the library and should always be initialized as a NULL pointer.

lpszPassword

A null-terminated string which specifies the password needed to access the certificate. Currently this member is only used if the CREDENTIAL_STORE_FILENAME option has been specified. If there is no password associated with the certificate, then this member should be initialized as a NULL pointer.

lpszCertStore

A null-terminated string which specifies the name of the certificate store to open. A certificate store is a collection of certificates and their private keys, typically organized by how they are used. If this value is NULL or points to an empty string, the default certificate store "MY" will be used.

Store Name	Description
CA	Certification authority certificates. These are certificates that are issued by entities which are entrusted to issue certificates to other individuals or organizations. Companies such as Comodo and DigiCert act as certification authorities.
MY	Personal certificates and their associated private keys for the current user. This store typically holds the client certificates used to establish a user's credentials. This corresponds to the "Personal" store that is displayed by the certificate manager utility and is the default store used by the library.
ROOT	Certificates that have been self-signed by a certificate authority. Root certificates for a number of different certification authorities such as Comodo and DigiCert are installed as part of the operating system and periodically updated by Microsoft.

lpszCertName

A null-terminated string which specifies the name of the certificate to use. The library will first search the certificate store for a certificate with a matching "friendly name"; this is a name for the certificate that is assigned by the user. Note that the name must match completely, but the comparison is not case sensitive. If no matching certificate is found, the library will then attempt to find a certificate that has a matching common name (also called the certificate subject). This comparison is less stringent, and the first partial match will be returned. If this second search fails, the library will return an error indicating that the certificate could not be found. If the SECURITY_PROTOCOL_SSH protocol has been specified, this member should be NULL.

lpszKeyFile

This member is not used and should always be NULL. The private key cannot be specified separately and must be associated with the certificate.

Remarks

An application only needs to create this structure if the server requires the client to provide a certificate as part of the process of negotiating the secure session. If a certificate is required, note that it must have a private key associated with it. Attempting to use a certificate that does not have a private key will result in an error during the connection process indicating that the client credentials could not be established.

If you are using a local certificate store, with the certificate and private key stored in the registry, you can explicitly specify whether the certificate store for the current user or the local machine (all users) should be used. This is done by prefixing the certificate store name with "HKCU:" for the current user, or "HKLM:" for the local machine. For example, a certificate store name of "HKLM:MY" would specify the personal certificate store for the local machine, rather than the current user. If neither prefix is

specified, then it will default to the certificate store for the current user. You can manage these certificates using the CertMgr.msc Microsoft Management Console (MMC) snap-in.

It is possible to load the certificate from a file rather than from current user's certificate store. The ***dwOptions*** member should be set to CREDENTIAL_STORE_FILENAME and the ***lpzCertStore*** member should specify the name of the file that contains the certificate and its private key. The file must be in Private Information Exchange (PFX) format, also known as PKCS #12. These certificate files typically have an extension of .pfx or .p12. Note that if a password was specified when the certificate file was created, it must be provided in the ***lpzPassword*** member or the library will be unable to access the certificate.

Note that the ***lpzUserName*** and ***lpzPassword*** members are values which are used to access the certificate store or private key file. They are not the credentials which are used when establishing the connection with the remote host or authenticating the client session.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: ctools12.h

Unicode: Implemented as Unicode and ANSI versions

The **SECURITYINFO** structure contains security related information about the connection, including the protocol, cipher suite and certificate.

```
typedef struct _SECURITYINFO
{
    DWORD        dwSize;
    DWORD        dwProtocol;
    DWORD        dwCipher;
    DWORD        dwCipherStrength;
    DWORD        dwHash;
    DWORD        dwHashStrength;
    DWORD        dwKeyExchange;
    DWORD        dwCertStatus;
    SYSTEMTIME    stCertIssued;
    SYSTEMTIME    stCertExpires;
    LPCTSTR       lpszCertIssuer;
    LPCTSTR       lpszCertSubject;
    LPCTSTR       lpszFingerprint;
} SECURITYINFO, *LPSECURITYINFO;
```

Members

dwSize

Specifies the size of the data structure. This member must always be initialized to the size of the structure in bytes prior to passing the address of this structure to the function. If this this member is not initialized, an error will be returned indicating that an invalid parameter has been passed to the function.

dwProtocol

A numeric value which specifies the security protocol selected when establishing the connection.

Value	Description
SECURITY_PROTOCOL_NONE	The connection is not secure. This value indicates a standard connection was established and all other structure members will have a value of zero.
SECURITY_PROTOCOL_TLS10	The TLS 1.0 protocol was selected when establishing a secure connection. This version is supported on all Windows platforms, however some servers may reject connections using version 1.0 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.
SECURITY_PROTOCOL_TLS11	The TLS 1.1 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. Some servers may reject connections using version 1.1 in favor of using version 1.2. This version should only be used for backwards compatibility with older servers that have not been updated to use the current version of TLS.

SECURITY_PROTOCOL_TLS12	The TLS 1.2 protocol was selected when establishing a secure connection. This version is supported on Windows 7 and later desktop platforms, and Windows Server 2008 R2 and later server platforms. This version of TLS offers the broadest compatibility with most servers.
SECURITY_PROTOCOL_TLS13	The TLS 1.3 protocol was selected when establishing a secure connection. This is the newest version of the protocol and is only supported on Windows 11, Windows Server 2022 and later versions of Windows. If this version is not supported by the operating system, TLS 1.2 will be used instead.

dwCipher

A numeric value which specifies the cipher that was selected when establishing the secure connection.

Value	Description
SECURITY_CIPHER_AES	The Advanced Encryption Standard cipher is a fixed block size cipher which use a key size of 128, 192 or 256 bits.
SECURITY_CIPHER_DES	The Data Encryption Standard block cipher was selected. This is a fixed key length cipher, using 56-bit keys.
SECURITY_CIPHER_DES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively providing a 168-bit length key.
SECURITY_CIPHER_DESX	A variant of the DES block cipher which XORs an extra 64-bits of the key before and after the plaintext has been encrypted, increasing the key size to 184 bits.
SECURITY_CIPHER_RC4	The RC4 stream cipher was selected. This is a variable key length cipher which supports keys between 40 and 128 bits, in 8-bit increments.
SECURITY_CIPHER_RC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.

dwCipherStrength

A numeric value which specifies the strength (the length of the cipher key in bits) of the cipher that was selected. Typically this value will be 128 or 256. 40-bit and 56-bit key lengths are considered weak encryption, and subject to brute force attacks. 128-bit and 256-bit key lengths are considered to be secure, and are the recommended key length for secure communications.

dwHash

A numeric value which specifies the hash algorithm which was selected.

Value	Description
SECURITY_HASH_MD5	The MD5 algorithm was selected. Its use has been deprecated and is no longer considered to be cryptographically secure.
SECURITY_HASH_SHA1	The SHA-1 algorithm was selected. Its use has been deprecated and is no longer considered to be

	cryptographically secure.
SECURITY_HASH_SHA256	The SHA-256 algorithm was selected.
SECURITY_HASH_SHA384	The SHA-384 algorithm was selected.
SECURITY_HASH_SHA512	The SHA-512 algorithm was selected.

dwHashStrength

A numeric value which specifies the strength (the length in bits) of the message digest that was selected.

dwKeyExchange

A numeric value which specifies the key exchange algorithm which was selected.

Value	Description
SECURITY_KEYEX_RSA	The RSA public key algorithm was selected.
SECURITY_KEYEX_KEA	The Key Exchange Algorithm (KEA) was selected. This is an improved version of the Diffie-Hellman public key algorithm.
SECURITY_KEYEX_DH	The Diffie-Hellman key exchange algorithm was selected.
SECURITY_KEYEX_ECDH	The Elliptic Curve Diffie-Hellman key exchange algorithm was selected. This is a variant of the Diffie-Hellman algorithm which uses elliptic curve cryptography.

dwCertStatus

A numeric value which specifies the status of the certificate returned by the secure server.

Value	Description
SECURITY_CERTIFICATE_VALID	The certificate is valid.
SECURITY_CERTIFICATE_NOMATCH	The certificate is valid, but the domain name does not match the common name in the certificate.
SECURITY_CERTIFICATE_EXPIRED	The certificate is valid, but has expired.
SECURITY_CERTIFICATE_REVOKED	The certificate has been revoked and is no longer valid.
SECURITY_CERTIFICATE_UNTRUSTED	The certificate or certificate authority is not trusted on the local system.
SECURITY_CERTIFICATE_INVALID	The certificate is invalid. This typically indicates that the internal structure of the certificate has been damaged.

stCertIssued

A structure which contains the date and time that the certificate was issued by the certificate authority. If the issue date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

stCertExpires

A structure which contains the date and time that the certificate expires. If the expiration date cannot be determined for the certificate, the SYSTEMTIME structure members will have zero values.

lpszCertIssuer

A null-terminated string which contains information about the organization that issued the certificate. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate issuer could not be determined, this member may be NULL.

lpzCertSubject

A null-terminated string which contains information about the organization that the certificate was issued to. This string consists of one or more comma-separated tokens. Each token consists of an identifier, followed by an equal sign and a value. If the certificate subject could not be determined, this member may be NULL.

lpzFingerprint

A null-terminated string which contains a sequence of hexadecimal values that uniquely identify the server. This member may be NULL.

Remarks

The size of this structure is typically determined using the **sizeof** operator in C/C++ and most other languages have a similar method for obtaining the size of a data structure or user-defined type. It is important to note the size of this structure will be different on 32-bit and 64-bit platforms due to the difference in the length of the structure members which are pointers.

When evaluating structure members which are string pointers, such as ***lpzCertIssuer*** and ***lpzCertSubject***, you should always check to ensure their values are not NULL (zero) before attempting to dereference them. If those values are undefined, it is not guaranteed they will point to zero-length strings.

There are additional protocols and ciphers defined which are not listed above because they are considered insecure and are no longer supported. It is important to note that most servers will reject attempts to use unsupported protocols such as SSL 3.0 and will either abort the connection or issue an error message and then immediately disconnect.

Requirements

Minimum Desktop Platform: Windows 7 Service Pack 1

Minimum Server Platform: Windows Server 2008 R2 Service Pack 1

Header File: ctools12.h

Unicode: Implemented as Unicode and ANSI versions

SocketWrench Library Error Codes

Value	Constant	Description
0x80042711	ST_ERROR_NOT_HANDLE_OWNER	Handle not owned by the current thread
0x80042712	ST_ERROR_FILE_NOT_FOUND	The specified file or directory does not exist
0x80042713	ST_ERROR_FILE_NOT_CREATED	The specified file could not be created
0x80042714	ST_ERROR_OPERATION_CANCELED	The blocking operation has been canceled
0x80042715	ST_ERROR_INVALID_FILE_TYPE	The specified file is a block or character device, not a regular file
0x80042716	ST_ERROR_INVALID_DEVICE	The specified device or address does not exist
0x80042717	ST_ERROR_TOO_MANY_PARAMETERS	The maximum number of function parameters has been exceeded
0x80042718	ST_ERROR_INVALID_FILE_NAME	The specified file name contains invalid characters or is too long
0x80042719	ST_ERROR_INVALID_FILE_HANDLE	Invalid file handle passed to function
0x8004271A	ST_ERROR_FILE_READ_FAILED	Unable to read data from the specified file
0x8004271B	ST_ERROR_FILE_WRITE_FAILED	Unable to write data to the specified file
0x8004271C	ST_ERROR_OUT_OF_MEMORY	Out of memory
0x8004271D	ST_ERROR_ACCESS_DENIED	Access denied
0x8004271E	ST_ERROR_INVALID_PARAMETER	Invalid argument passed to function
0x8004271F	ST_ERROR_CLIPBOARD_UNAVAILABLE	The system clipboard is currently unavailable
0x80042720	ST_ERROR_CLIPBOARD_EMPTY	The system clipboard is empty or does not contain any text data
0x80042721	ST_ERROR_FILE_EMPTY	The specified file does not contain any data
0x80042722	ST_ERROR_FILE_EXISTS	The specified file already exists
0x80042723	ST_ERROR_END_OF_FILE	End of file
0x80042724	ST_ERROR_DEVICE_NOT_FOUND	The specified device could not be found
0x80042725	ST_ERROR_DIRECTORY_NOT_FOUND	The specified directory could not be found
0x80042726	ST_ERROR_INVALID_BUFFER	Invalid memory address passed to function
0x80042728	ST_ERROR_NO_HANDLES	No more handles available to this process
0x80042733	ST_ERROR_OPERATION_WOULD_BLOCK	The specified operation would block the current thread
0x80042734	ST_ERROR_OPERATION_IN_PROGRESS	A blocking operation is currently in progress
0x80042735	ST_ERROR_ALREADY_IN_PROGRESS	The specified operation is already in progress
0x80042736	ST_ERROR_INVALID_HANDLE	Invalid handle passed to function
0x80042737	ST_ERROR_INVALID_ADDRESS	Invalid network address specified

0x80042738	ST_ERROR_INVALID_SIZE	Datagram is too large to fit in specified buffer
0x80042739	ST_ERROR_INVALID_PROTOCOL	Invalid network protocol specified
0x8004273A	ST_ERROR_PROTOCOL_NOT_AVAILABLE	The specified network protocol is not available
0x8004273B	ST_ERROR_PROTOCOL_NOT_SUPPORTED	The specified protocol is not supported
0x8004273C	ST_ERROR_SOCKET_NOT_SUPPORTED	The specified socket type is not supported
0x8004273D	ST_ERROR_INVALID_OPTION	The specified option is invalid
0x8004273E	ST_ERROR_PROTOCOL_FAMILY	Specified protocol family is not supported
0x8004273F	ST_ERROR_PROTOCOL_ADDRESS	The specified address is invalid for this protocol family
0x80042740	ST_ERROR_ADDRESS_IN_USE	The specified address is in use by another process
0x80042741	ST_ERROR_ADDRESS_UNAVAILABLE	The specified address cannot be assigned
0x80042742	ST_ERROR_NETWORK_UNAVAILABLE	The networking subsystem is unavailable
0x80042743	ST_ERROR_NETWORK_UNREACHABLE	The specified network is unreachable
0x80042744	ST_ERROR_NETWORK_RESET	Network dropped connection on remote reset
0x80042745	ST_ERROR_CONNECTION_ABORTED	Connection was aborted due to timeout or other failure
0x80042746	ST_ERROR_CONNECTION_RESET	Connection was reset by remote network
0x80042747	ST_ERROR_OUT_OF_BUFFERS	No buffer space is available
0x80042748	ST_ERROR_ALREADY_CONNECTED	Connection already established with remote host
0x80042749	ST_ERROR_NOT_CONNECTED	No connection established with remote host
0x8004274A	ST_ERROR_CONNECTION_SHUTDOWN	Unable to send or receive data after connection shutdown
0x8004274C	ST_ERROR_OPERATION_TIMEOUT	The specified operation has timed out
0x8004274D	ST_ERROR_CONNECTION_REFUSED	The connection has been refused by the remote host
0x80042750	ST_ERROR_HOST_UNAVAILABLE	The specified host is unavailable
0x80042751	ST_ERROR_HOST_UNREACHABLE	Remote host is unreachable
0x80042753	ST_ERROR_TOO_MANY_PROCESSES	Too many processes are using the networking subsystem
0x8004276B	ST_ERROR_NETWORK_NOT_READY	Network subsystem is not ready for communication
0x8004276C	ST_ERROR_INVALID_VERSION	This version of the operating system is not supported
0x8004276D	ST_ERROR_NETWORK_NOT_INITIALIZED	The networking subsystem has not been initialized
0x80042775	ST_ERROR_REMOTE_SHUTDOWN	The remote host has initiated a graceful shutdown sequence

0x80042AF9	ST_ERROR_INVALID_HOSTNAME	The specified hostname is invalid or could not be resolved
0x80042AFA	ST_ERROR_HOSTNAME_NOT_FOUND	The specified hostname could not be found
0x80042AFB	ST_ERROR_HOSTNAME_REFUSED	Unable to resolve hostname, request refused
0x80042AFC	ST_ERROR_HOSTNAME_NOT_RESOLVED	Unable to resolve hostname, no address for specified host
0x80042EE1	ST_ERROR_INVALID_LICENSE	The license for this product is invalid
0x80042EE2	ST_ERROR_PRODUCT_NOT_LICENSED	This product is not licensed to perform this operation
0x80042EE3	ST_ERROR_NOT_IMPLEMENTED	This function has not been implemented on this platform
0x80042EE4	ST_ERROR_UNKNOWN_LOCALHOST	Unable to determine local host name
0x80042EE5	ST_ERROR_INVALID_HOSTADDRESS	Invalid host address specified
0x80042EE6	ST_ERROR_INVALID_SERVICE_PORT	Invalid service port number specified
0x80042EE7	ST_ERROR_INVALID_SERVICE_NAME	Invalid or unknown service name specified
0x80042EE8	ST_ERROR_INVALID_EVENTID	Invalid event identifier specified
0x80042EE9	ST_ERROR_OPERATION_NOT_BLOCKING	No blocking operation in progress on this socket
0x80042F45	ST_ERROR_SECURITY_NOT_INITIALIZED	Unable to initialize security interface for this process
0x80042F46	ST_ERROR_SECURITY_CONTEXT	Unable to establish security context for this session
0x80042F47	ST_ERROR_SECURITY_CREDENTIALS	Unable to open client certificate store or establish client credentials
0x80042F48	ST_ERROR_SECURITY_CERTIFICATE	Unable to validate the certificate chain for this session
0x80042F49	ST_ERROR_SECURITY_DECRYPTION	Unable to decrypt data stream
0x80042F4A	ST_ERROR_SECURITY_ENCRYPTION	Unable to encrypt data stream
0x80043031	ST_ERROR_MAXIMUM_CONNECTIONS	The maximum number of client connections exceeded
0x80043032	ST_ERROR_THREAD_CREATION_FAILED	Unable to create a new thread for the current process
0x80043033	ST_ERROR_INVALID_THREAD_HANDLE	The specified thread handle is no longer valid
0x80043034	ST_ERROR_THREAD_TERMINATED	The specified thread has been terminated
0x80043035	ST_ERROR_THREAD_DEADLOCK	The operation would result in the current thread becoming deadlocked
0x80043036	ST_ERROR_INVALID_CLIENT_MONIKER	The specified moniker is not associated with any client session
0x80043037	ST_ERROR_CLIENT_MONIKER_EXISTS	The specified moniker has been assigned to another client session

0x80043038	ST_ERROR_SERVER_INACTIVE	The specified server is not listening for client connections
0x80043039	ST_ERROR_SERVER_SUSPENDED	The specified server is suspended and not accepting client connections