Developer's Guide and Technical Reference
Version 11.0.2218.1855

Introduction

The SocketTools ActiveX Edition includes ActiveX controls (OCXs) which can be used with Visual Basic 6.0, Visual FoxPro, Visual Basic for Applications (VBA) and other legacy development languages with support the ActiveX standard. The ActiveX Edition is ideal for the developer who requires the flexibility, ease of use and rapid development features of a component without the complexities of working with the Windows Sockets API or in-depth knowledge of how the various Internet protocols are implemented. The SocketTools ActiveX Edition consists of fourteen core networking components which can be used to develop applications which meet a wide range of needs. SocketTools covers it all, including uploading and downloading files, sending and retrieving email, remote command execution, terminal emulation, and much more.

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

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

The following are just ome of the features in the SocketTools 11 ActiveX Edition:

- ATL based ActiveX controls with no additional runtime library dependencies
- Support for Windows 11 and Windows Server 2025
- Includes both high level and lower level interfaces for maximum flexibility
- Support for both synchronous and asynchronous network connections
- Includes controls that can be used to create custom client and server applications
- Provides cloud-based application storage and geographical IP location services
- Support for the TLS 1.2 and TLS 1.3 protocols with 256-bit AES encryption
- Support for both implicit and explicit TLS connections
- Support for the SSH protocol and integrated support for SFTP as part of the FTP control
- Support for standard and secure proxy servers using FTP and HTTP
- Support for using client and server certificates in PKCS #12 format
- Thread-safe implementation with full support for multithreaded applications
- An extensive Developer's Guide and online Technical Reference
- Easy redistribution for any number of applications and end users

Developer's Guide

To help you get started using SocketTools, the Developer's Guide covers a variety of programming topics related to SocketTools, as well an overview of each of the controls included in the product. Even if you have experience working with previous versions of SocketTools, we recommend that you review the Developer's Guide. If you are using a language other than Visual Basic, 6.0 you'll also find some very helpful information about how to make the most of SocketTools in other programming languages such as Visual C++.

Technical Reference

The Technical Reference provides extensive documentation on all of the functions in each of the SocketTools controls. It's here that you'll find information on the various properties, methods and events provided by the component. If it is your first time using a particular control, we recommend that you first read the overview of

that control in the Developer's Guide.

SocketTools Licensing Information

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

Evaluation Licenses

When you install SocketTools, you are given the option of entering a serial number or proceeding with the installation without a serial number. If you install SocketTools 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 SocketTools components may not be redistributed to third-parties. After the evaluation period has ended, you must either purchase a development license or remove SocketTools from your computer system.

Runtime Licensing

Most languages which use ActiveX 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 a 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 call the Initialize method.

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

If you install SocketTools 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 SocketTools controls.

License Manager

Included with your copy of SocketTools 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 SocketTools 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.

SocketTools 11 Upgrade Information

This section will help you upgrade an application written using a previous version of the SocketTools ActiveX Edition. 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

SocketTools 11 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 SocketTools. We recommend using the lasted 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 SocketTools 11 ActiveX controls 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. Although the ActiveX controls may be used with Visual Basic .NET and Visual C#, it is recommended you use the SocketTools .NET Edition if you are creating applications for the .NET Framework.

The Visual Basic 6.0 runtime is supported on Windows 11, however the IDE is no longer supported by Microsoft. To use the Visual Basic 6.0 IDE in Windows 11, you will need to run it with elevated privileges. Visual Studio 2022 supports the use of ActiveX controls, although there can be significant limitations and performance implications when using ActiveX components in .NET. If you are migrating an existing legacy project to a modern version of .NET, we recommend you also switch to the SocketTools .NET Edition which provides native .NET components.

Upgrading Projects

If you are upgrading from version 10, applications will be source code compatible with the SocketTools 11 ActiveX controls. In most cases, all you will need to do is install the current version, update the control references in your project and recompile your application. It is important to note that the control file names, class IDs and interfaces have changed and are not binary compatible with previous versions.

If you are upgrading from an earlier version of SocketTools, you should find that most projects which have used SocketTools 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 SocketTools, you may find that there have been changes to method names and parameters which will need to be updated.

The runtime license key has changed for SocketTools 11, which may require you to redefine the value in your application when calling the control's **Initialize** method. As with previous versions of SocketTools, 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 11 controls and an error will be returned if an invalid runtime key is specified.

Your application should not attempt to reference the current version of a control and an earlier version of the control within the same application. When upgrading to version 11, first remove all references to the earlier version of the control, save the project and reload it. Then add the reference to the version 11 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.11" and not simply "SocketTools.FtpClient". If the major version number is omitted, the latest version of the control will always be loaded.

With SocketTools 11, secure connections will use TLS 1.2 or later. By default, the controls 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 that 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 the controls to use earlier versions of TLS for backwards compatibility with older servers. This is done by explicitly setting the **SecureProtocol** property to specify the protocol version required. However, this is not generally recommended because using SSL 3.0 or TLS 1.0 may cause servers to immediately reject the connection attempt.

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

Control 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. For more information, refer to the Redistribution section.

File Name	ProgID	Description
csdnsx11.ocx	SocketTools.DnsClient.11	Domain Name Service Control
csftpx11.ocx	SocketTools.FtpClient.11	File Transfer Protocol Control
csftsx11.ocx	SocketTools.FtpServer.11	File Transfer Server Control
cshtpx11.ocx	SocketTools.HttpClient.11	Hypertext Transfer Protocol Control
cshtsx11.ocx	SocketTools.HttpServer.11	Hypertext Transfer Server Control
csicmx11.ocx	SocketTools.IcmpClient.11	Internet Control Message Protocol Control
csmapx11.ocx	SocketTools.ImapClient.11	Internet Message Access Protocol Control
csmsgx11.ocx	SocketTools.MailMessage.11	Mail Message Control
csmtpx11.ocx	SocketTools.SmtpClient.11	Simple Mail Transfer Protocol Control
csncdx11.ocx	SocketTools.FileEncoder.11	File Encoding Control
csnvtx11.ocx	SocketTools.Terminal.11	Terminal Emulation Control
csnwsx11.ocx	SocketTools.NntpClient.11	Network News Transfer Protocol Control
cspopx11.ocx	SocketTools.PopClient.11	Post Office Protocol Control
csrasx11.ocx	SocketTools.Dialer.11	Remote Access Services Dialer Control
csrshx11.ocx	SocketTools.RshClient.11	Remote Command Protocol Control
csrssx11.ocx	SocketTools.NewsFeed.11	Syndicated News Feed Control
cstimx11.ocx	SocketTools.TimeClient.11	Time Protocol Control
cstntx11.ocx	SocketTools.TelnetClient.11	Telnet Protocol Control

cstshx11.ocx	SocketTools.SshClient.11	Secure Shell Protocol Control
cstxtx11.ocx	SocketTools.TextMessage.11	Text Messaging Control
cswebx11.ocx	SocketTools.WebStorage.11	Web Storage Control
cswhox11.ocx	SocketTools.WhoisClient.11	Whois Protocol Control
cswipx11.ocx	SocketTools.WebLocation.11	Web Location Control
cswskx11.ocx	SocketTools.SocketWrench.11	Windows Sockets (SocketWrench) Control
cswsvx11.ocx	SocketTools.InternetServer.11	Windows Sockets Server Control

SocketTools Product Evaluation

If you install SocketTools 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 SocketTools controls 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 SocketTools, you cannot redistribute the controls 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 SocketTools 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 SocketTools 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.

SocketTools Component Redistribution

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

System Requirements

SocketTools 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 2000, Windows XP and Windows Vista are no longer supported. SocketTools 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, SocketTools components will continue to function on those platforms.

Control Redistribution

For those applications created using the SocketTools ActiveX controls, the control 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 provide 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 installation tools are capable of automatically handling control registration. For custom installations, refer to the article Self Registration for Controls for information on how ActiveX controls implement self-registering.

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. This utility accepts a command line argument which specifies the name of the control to register. For example, the following command would register the 32-bit SocketTools FtpClient control on a 64-bit Windows system:

```
%WINDIR%\SysWOW64\regsvr32 %WINDIR%\SysWOW64\csftpx11.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 because it modifies the system registry. To register controls from the command prompt, you must run it with administrative privileges. If a registration error occurs, it is likely because you are attempting to register the control without the appropriate privileges.

If the ActiveX controls are installed on a 64-bit version of Windows, the 32-bit controls, which are used with Visual Basic 6.0 and other 32-bit development tools, are installed in the C:\Windows\SysWOW64 folder and the 64-bit controls are installed in the C:\Windows\System32 folder. When redistributing the ActiveX controls, 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 controls, 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.

Version Information

The SocketTools controls have embedded information which provides version information to an installation utility. This information, called the version resource, specifies the 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 control, the setup program must determine if it has already been installed on the target system. If it has, it must compare the version resource information in the two files. It should only overwrite the control OCX file 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 file without checking the version number may cause other programs to fail unexpectedly on the end-user's system, which is obviously not desirable.

Installation Directory

The SocketTools ActiveX controls should typically be installed in the C:\Windows\System32 folder on the local machine. If you are deploying to a 64-bit system, then the 32-bit controls should be installed in the C:\Windows\SysWOW64 folder. 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.

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 controls are installed and registered in the correct location. Consult the documentation for your installer to determine if it is 64-bit compatible.

Windows Install Packages

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

Package Name	Description
cstools11_activex_x86.msi	SocketTools 11 redistributable ActiveX controls for 32-bit applications. This installer is what most developers would use, particularly if the application was created using Visual Basic 6.0. It can be installed on both 32-bit and 64-bit

	versions of Windows.
cstools11_activex_x64.msi	SocketTools 11 redistributable ActiveX controls for 64-bit applications. This installer should only be used if 64-bit development tools were used, and can only be installed on 64-bit versions of Windows.

If you're redistributing a 32-bit application, then all you need is the x86 installer package. If you're redistributing a 64-bit application, then you need the x64 installer package. If you're developing with Visual Basic 6.0, you should only use the x86 installer. The installer packages will make sure the SocketTools controls are installed in the correct location and will perform the appropriate version checking.

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

```
msiexec /qn /I cstools11_activex_x86.msi
```

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

Technical Support

Catalyst Development 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 Documentation

A comprehensive collection of online help 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.

Release Notes

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

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.

Premium Support

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

License Agreement

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

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

1. GRANT OF LICENSE

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

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

2. COPYRIGHT

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

3. REDISTRIBUTION

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

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

(c) If you redistribute the sample code or redistributable components, you agree to: (i) distribute the redistributables in object code only, in conjunction with and as a part of a software application product developed by you which adds significant and primary functionality to the Software; (ii) not use Catalyst Development's name, logo, or trademarks to market your software application product; (iii) include a valid copyright notice on your software product ; (iv) indemnify, hold harmless, and defend Catalyst Development from and against any claims or lawsuits, including attorney's fees, that arise or result from the use or

distribution of your software application product; (v) not permit further distribution of the redistributables by your end user.

4. UPGRADES

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

5. LICENSE RESTRICTIONS

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

6. CONFIDENTIALITY

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

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

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

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

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

7. CONTINUATION OF SERVICE

Some features of the Software may require the use of remote servers under the control of Catalyst

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

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

8. LIMITED WARRANTY

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

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

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

9. LIMITATION OF LIABILITY

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

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

10. GOVERNMENT-RESTRICTED RIGHTS

United States Government Restricted Rights. The Software and related documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to the restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer for such purposes is Catalyst Development Corporation, 56925 Yucca Trail #254, Yucca Valley, CA 92284

11. EXPORT CONTROLS

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

12. PROHIBITED DESTINATIONS

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

13. GOVERNING LAW

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

14. GENERAL PROVISIONS

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

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

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

Copyright Information

SocketTools 11 ActiveX Edition Developers Guide

Internet Control Message Protoco (ICMP) Control

- Internet Message Access Protocol (IMAP) Control
- Internet Server Control
- Mail Message (MIME) Control
- Network News Transfer Protocol (NNTP) Control
- News Feed (RSS) Control
- Post Office Protocol (POP3) Control
- Remote Access Services (RAS) Control
- Remote Command Protocol (RSH) Control
- Secure Shell Protocol (SSH) Control
- Simple Mail Transfer Protocol (SMTP) Control
- SocketWrench (TCP/IP) Control
- Telnet Protocol Control
- Terminal Emulation Control
- Text Message Control
- Time Protocol Control
- Web Location Control
- Web Storage Control
- Whois Protocol Control

Introduction

The SocketTools ActiveX Edition is a collection of ActiveX components (OCXs) which provide application programming interfaces for a variety of Internet protocols. Using SocketTools, you can quickly and easily write programs which upload and download files, send and retrieve email messages, execute commands on servers, establish virtual terminal sessions and perform many other tasks. With the SocketTools ActiveX Edition it is not required that you be a networking expert or understand the details of how certain networking or application protocols work.

The Developer's Guide will cover the basics of Internet programming in general, and then move on to the specific protocols supported by the SocketTools ActiveX Edition. When references are made to the Internet, the same information can generally be applied to corporate intranets. For example, the SocketTools Hypertext Transfer Protocol component can be used to communicate with a company's internal web server as well as web servers over the Internet.

SocketTools Features

The SocketTools ActiveX controls can be used in a wide variety of programming languages, including Visual Basic 6.0, Visual Basic for Applications (VBA), Visual FoxPro and a variety of other development environments. Most languages which are capable of using ActiveX controls and/or COM objects can use the controls in the SocketTools ActiveX Edition.

Features of the SocketTools ActiveX Edition include:

- The ActiveX controls provide a simple interface that is easy to use and understand. There are no complicated function calls, and most methods support optional arguments that only need to be specified if required by the application. Most complex operations can be performed with only a few lines of code.
- There are no external dependencies on third party libraries or components, and each ActiveX component is completely self-contained. We do not require that you redistribute large shared libraries like the Microsoft Foundation Classes or Visual C++ runtime libraries. Not only does this make redistribution of your software easier, it can reduce the overall footprint for applications which do not need to use these libraries themselves.
- The controls provide broad-based compatibility with a variety of programming languages, including scripting languages such as VBScript. Methods and events are designed to use variant data types to ensure a high degree of compatibility with all development tools.
- A comprehensive design which supports both high-level operations as well as lower-level methods at the protocol level. For example, the File Transfer Protocol component has methods such as **PutFile** and **GetFile** which allow an application to easily upload and download files in a single method call. It also includes lower-level methods like **OpenFile** to open a file on the server and access it in a fashion similar to traditional file I/O operations.
- Support for both synchronous (blocking) and asynchronous (non-blocking) operation depending on the needs of the application. Asynchronous operation is supported by an event-driven model where the application is notified of networking events by events generated by the component. Event notification can be enabled, disabled and resumed completely under the control of the application, giving developers complete freedom in controlling their behavior of their software. Synchronous operation is also fully supported, enabling developers to easily write programs without using an event-driven approach.
- The ActiveX Edition enables applications to take advantage of security features, such as support for the transport Layer Security (TLS) protocol and 256-bit encryption. The components use the Windows CryptoAPI to provide security services, which means there are no third-party security libraries that must be installed by your users. Taking advantage of the security features in the SocketTools ActiveX Edition is as simple as setting a few properties before connecting to the server. The protocol negotiation, data encryption and decryption is handled transparently by the control. From the perspective of the application developer, it is just as if it were a standard connection to the server.

The SocketTools ActiveX Edition includes everything professional software developers need to create complex programs that take advantage of the standard Internet protocols, enabling developers to focus on their core application technology rather than the details of how a particular protocol is implemented or understanding the specifics of Windows Sockets programming.

Getting Started

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

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

File Transfers

- File Transfer Protocol
- Hypertext Transfer Protocol

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

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

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

World Wide Web

- Hypertext Transfer Protocol

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

Electronic Mail

- Domain Name Services Protocol
- Internet Message Access Protocol
-

Mail Message Library
- Post Office Protocol
- Simple Mail Transfer Protocol

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

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

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

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

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

Terminal Sessions

- Rlogin Protocol
- Telnet Protocol
- Terminal Emulation

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

Newsgroups

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

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

General Concepts

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

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 the Windows Sockets API to exchange information over the Internet or a local intranet. One is to code directly against the API which requires an in-depth understanding of the sockets interface and the application protocol being used. The other approach is to use a component which provides a higher-level interface that implements the various protocols by setting properties, calling methods and responding to events. This can provide a more natural programming interface for many languages, and it allows you to avoid many of the complex issues associated with network programming. By simply 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 design of the SocketTools components in general, the learning curve is low and experimentation is easy.

Application Protocols

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

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

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

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

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

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 mixed 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 establishes 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 terminated.

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

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.

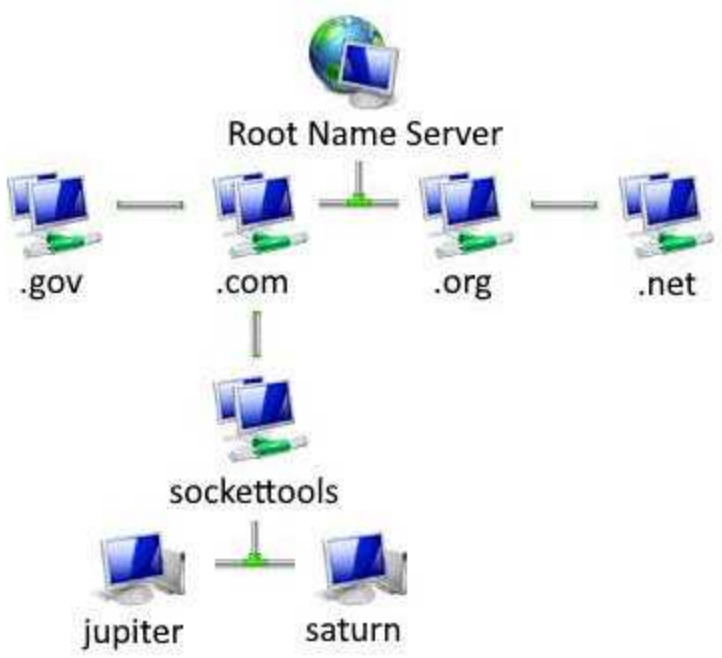
UDP is sometimes referred to as an unreliable protocol because when a program sends a UDP datagram over the network, there is no way for it to know that it actually arrived at its destination. This means that the sender and receiver must typically implement their own application protocol on top of UDP. 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 insure 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.

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

Domain Names

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.

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

Service Ports

In addition to the IP address of the server, 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 number between 1 and 65535 that uniquely identifies an application running on the system. A port can be referred to by its number, or by a name that is associated with that number. 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 used for transferring files, accessing documents on a webserver or sending and receiving email messages. In most cases, when connecting to a service using the SocketTools libraries, they will default to the appropriate port number for that server. For example, the File Transfer Protocol control has default port values for standard and secure connections. Specifying a different port number is only necessary if you know that the server has been configured to use a non-standard port number.

It is important to remember that a service name or port number is a way to address an application running on a server. 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 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 a group of related protocols. 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.

For the most part, it is not necessary for applications which use the SocketTools controls to directly make use of the low-level socket interface in order to communicate over the Internet. Instead, SocketTools provides a higher level of abstraction where a connection is managed through the control interface.

Security Protocols

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. The SocketTools ActiveX Edition has the ability to establish secure connections with servers. Although most of the technical issues such as data encryption are handled internally by the control, 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 the individual or company which is issued a certificate are 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 have resulted in the Transport Layer Security (TLS) protocol, and it has become the 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.

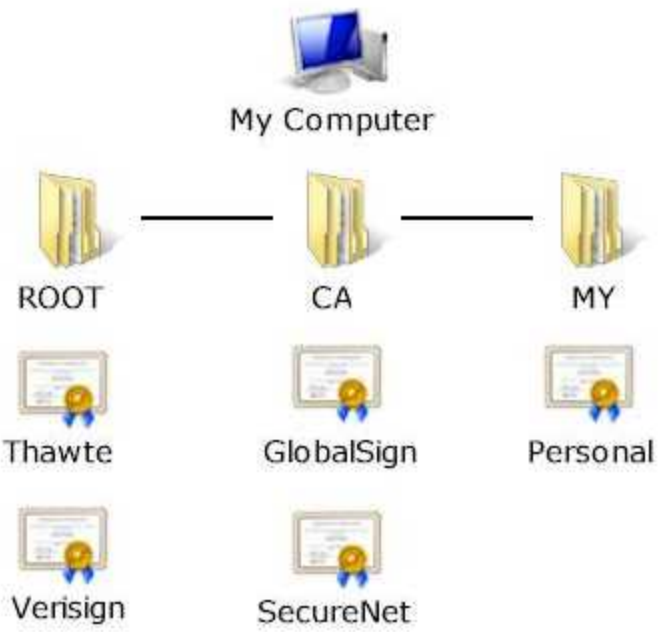
A secure connection, for example between a web browser and a server, begins with what is called the handshake phase where the client and server identify themselves. When the client first connects with the server it sends a block of data to the server and the server responds with its digital certificate, 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. SocketTools 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 to whom the certificate was issued (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 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 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 or DigiCert. That organization 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 connects to their server and checks the certificate, they see that it was issued by a trusted Certification Authority. In essence, the user is saying that because they trust the Certificate Authority, and the Certificate Authority trusts the company to whom the certificate was issued, 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.exe 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 or DigiCert. 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 SocketTools.

SocketTools Development

The SocketTools ActiveX Edition provides a comprehensive collection of ActiveX controls for performing a variety of Internet related programming tasks. Although the number of properties, methods and events may appear daunting, once you begin using SocketTools in your own applications you'll find that the various controls are designed to work together in a cohesive fashion. After you've familiarized yourself with one control, the others will become much simpler to use.

Throughout the Developer's Guide there are some general concepts and terminology used that are essential to understanding how SocketTools 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 SocketTools, 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. SocketTools uses a one-to-one relationship between an instance of a control and a client session. By creating multiple instances of a control, an application can create multiple client/server sessions if necessary.

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. For example, the File Transfer Protocol control establishes one connection, called the command channel, when the client initially connects to the server. However, when a file is being uploaded or downloaded, a second connection called the data channel is created just for that transfer. When the transfer completes, the second connection is terminated while the original command channel connection remains active. Even though there are multiple connections being made, SocketTools considers it to be a single client session. An active session is referenced by the instance of the control that was used to create the session. When the session is no longer needed, the control's Disconnect method will terminate the connection to the server and release the resources allocated for that session. After that point, the session is no longer valid and subsequent function calls using the control 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. SocketTools provides one of two general types of authentication methods, depending on the protocol. For protocols which require the client to authenticate itself, the controls will provide a Login method. For protocols where authentication is optional, the controls will provide an Authenticate method. Refer to the technical reference for the specific protocol to determine if authentication is required.

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 SocketTools 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 a connection event, which is generated whenever an asynchronous network connection is completed by the client. Another example is a progress event, which is generated periodically by the control to inform the client of its progress as it sends or receives data. To determine what events are available in a specific control, refer to the documentation. More specific information about event handling is provided later in this guide.

Application Design

The SocketTools ActiveX Edition 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 a control 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 Hypertext Transfer Protocol (HTTP) control which has a variety of high level methods such as **GetFile**, **PostData** and so on. Using these methods, your application can perform the most common tasks for that protocol with a minimum of coding. You don't need to even understand the basics of how the protocol works, or what the control is doing. The high level methods allow you to program against the control 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, the HTTP control also has methods like Command, which enable you to send custom commands to a web server. Normally, for operations like retrieving a file or posting data to a script, this isn't necessary. But if your application needs to use WebDAV, a set of extensions to the HTTP protocol to support distributed web authoring, then the lower level methods like Command enable you to do this.

If you are generally new to Internet programming or are just getting started with SocketTools, 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 control 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 a large toolkit like SocketTools 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 SocketTools controls work, expand on it.

Program

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

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

Initialization prepares the control 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 server. Finally, before your program terminates, you uninitialize the control which causes it to perform any necessary housekeeping prior to releasing any system resources which were allocated on behalf of your program.

Control Initialization

When you begin developing your application using one of the SocketTools controls, 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 SocketTools 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 SocketTools 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 SocketTools and then purchased a license, the license key can be created using the License Manager utility that was included with SocketTools. 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 SocketTools and is defined in a file named "csrtkey11" which can be included with your application. For example, C/C++ programmers would use the **csrtkey11.h** header file while Visual Basic programmers would use the **csrtkey11.bas** module. The Visual Basic module would define the runtime license key as:

```
'
' SocketTools 11 Build 2218
' Copyright 2025 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 CSTOOLS11_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 = ftpClient.Initialize(CSTOOLS11_LICENSE_KEY)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools 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 control that is used. If both the File Transfer Protocol and Hypertext Transfer Protocol controls were being used in the same application, it would be required to call the **Initialize** methods for each control at the beginning of the program.

It is safe to call the initialization method more than once, but for each time that it is called, you should 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 should 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.

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 server. 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 server 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 your 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 server 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 server. 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 server 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 your 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 insure that your program will work correctly, regardless of the speed or configuration of the network.

Secure Connections

The SocketTools ActiveX Edition supports the ability to create secure connections using the standard TLS 1.2 and 1.3 protocols. For those Internet protocols which support secure connections, 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 protocols have additional options that control how the secure session is established. Secure connections may either be implicit or explicit, depending on the protocol. An implicit 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, 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. SocketTools supports both implicit and explicit secure connections, and this is also controlled by the options provided to the Connect method.

In addition to establishing a secure connection, the client may be required to provide additional authentication information to the server in form a client certificate. A secure 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 for the control. Additional information about secure connections and certificates is also available in the Developer's Guide.

Network Input/Output

Each of the SocketTools networking controls provides methods for exchanging data between your application and the server. 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 pass byte arrays, not strings, to the Read and Write methods if possible.

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 server will receive that number of bytes in a single read. However, this is not how TCP works, and by extension, not how the SocketTools libraries work 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 sent 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:

1. 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.

2. 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.

3. 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 server that a complete command has been received.

4. 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 in the SocketTools controls. The higher level methods automatically handle the lower level network I/O for you. For example, the **GetData** method in the File Transfer Protocol control will retrieve a file from the server and return the entire contents to your application in a single method call. When using the high level methods, the details of how the data is read and processed is handled by the control and no additional coding is required on your part.

Event Handling

In SocketTools, event notification provides a mechanism for the component 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 server. 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 a file transfer.

Asynchronous network events require that the Blocking property for the control be set to False. This will set the control into a non-blocking mode. All of the networking controls share the same common set of events and all function in a similar way. These events are:

OnConnect

This event is generated whenever a connection to a server has completed. Unlike a blocking connection, when the control 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 is generated whenever the server 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 server closes its connection to you. It is also important to keep in mind that although the server 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 is generated whenever the server sends data to your application. Once this event has fired, it will not be triggered again until you read at least some of the data that has been sent to you. It is not recommended that any complex operations be performed in the **OnError** event handler. Applications should update any state variables or user interface objects and exit the handler immediately. Performing another operation using the control in an **OnError** handler can potentially result in the event handler being called recursively.

OnWrite

This event is generated whenever there is enough memory available in the local send buffers to accommodate some data. It is generated after a connection has completed, which tells your application that it may begin sending data to the server. It will also be generated if a call to the Write method fails with the error that it would cause the application to block. In this case, when the socket is able to send 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 server sends data to your application, signaling to you that there is data available to be read. Even though the server may continue sending you more data, another **OnRead** event will not be generated until you read at least some 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 file transfer using the File Transfer Protocol control. The most common status events are:

OnCommand

This event is used by the control to inform the application of the status of a command sent to the server. It applies to those protocols which use explicit commands to initiate actions. Examples would be the File Transfer Protocol (FTP) control, Hypertext Transfer Protocol (HTTP) control and Internet Message Access Protocol (IMAP) control. Note that the actual result codes returned by the servers depend on the specific protocol, and can also vary among various server implementations.

OnError

This event is used by the control 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.

OnProgress

This event is used by the control to inform the application of the progress of a blocking operation, such as a file transfer. Note that in some cases, the control may not be able to determine the total amount of data to be transferred, which would prevent a percentage from being calculated. For example, this can occur using the Hypertext Transfer Protocol (HTTP) control if the resource being downloaded is created dynamically on the server, such as an ASP page. In this case, because the server is unable to specify the total size of the resource, the control 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.

These 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 dialog 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, the **On Error** statement 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 numeric value. A value of zero indicates that the method completed successfully and that no error occurred. A non-zero return value specifies an error code which indicates the reason for the failure. For programmers who prefer to handle exceptions, rather than check return values for each method, the controls have a property called **ThrowError**. If set to True, then when a method fails it will throw an exception that must be caught by the application. Just as an error that occurs when setting a property, if the **ThrowError** property is set to true and an error occurs without there being an exception handler in place, the application will typically terminate.

To determine the error code for the last error generated by the control, 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.

Debugging Facilities

All of the SocketTools networking controls include 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 cstrcv11.dll library along with those SocketTools controls you are using in your program. The **cstrcv11.dll** library is what performs the actual logging and must be in a directory where it can be loaded by your application. It is recommended that you either install it in the Windows system directory or the directory where your application is installed. Note that this is a standard Windows dynamic link library and it does not need to be registered.

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 server 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, SocketTools will open the file, append the logging data and then close the file. The controls 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 one of the controls. 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 SocketTools controls, it is only necessary to enable logging in one of them. Once enabled, all SocketTools network operations in the current thread will be logged, regardless of which control has enabled logging.

Language Support

The SocketTools ActiveX Edition can be used with a wide variety of legacy programming languages and software development tools for Windows. To determine if your development language is capable of using the ActiveX Edition, it should support all of the following features:

- It needs to support the Component Object Model (COM) specification, and support the ability to create instances of a COM object. This is typically done in one of three ways: a function call to create an instance of the object, adding the object to a project and referencing it, or placing the object on a visual form or dialog.
- The language must provide support for variant data types. A variant is a special data type which can be used to represent multiple types of data, including integer, string, date and currency values. All of the SocketTools methods and events use variant types and the developer is responsible for converting those variants into the required data type. For example, in Visual C++, this can be accomplished using the **CComVariant** class.
- The language must support passing method parameters by value and by reference. When a variable is "passed by value", a copy of its value is passed to the method and the original value remains unchanged. However, when a variable is "passed by reference", the memory address of the variable (typically called a pointer) is passed to the method, enabling the method to modify its value. In most cases, this is handled transparently by the language. Note that some languages may require that you explicitly specify that a variable will be passed by reference using a specific keyword.
- The language must support event handlers which have variants passed by value as event parameters. If the language incorrectly assumes that all event parameters are passed by reference, this will prevent event notifications from working correctly. In general this is not an issue with any current languages, but may present a problem in older versions of a language. If you experience a problem with event handling in your language, contact the company to make sure that they are capable of correctly handling event notifications from an ActiveX component that passes parameters by value.

Microsoft Visual Basic, Visual FoxPro and VBScript are all examples of languages which can use the SocketTools ActiveX Edition. If your programming language is capable of using ActiveX controls or indicates that it supports OLE Automation, then you should be able to use SocketTools. Consult your language technical reference for additional information about how to create an instance of an ActiveX/COM object and reference its properties, methods and events.

Data Types

Because various languages handle data types in different ways, the SocketTools components have been designed to use variants. A variant is a special data type which can be used to represent multiple types of data, including integer, string, date and currency values. All of the SocketTools methods and events use variant types and the developer is responsible for converting those variants into the required data type. For example, in Visual C++, this can be accomplished using the CComVariant class.

In addition to variant types, the controls also use numeric data types for many property values. 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++	VB 6	VB.NET
Byte	1 byte	0 to 255	BYTE	Byte	Byte
Boolean	4 bytes	0 is False, 1 is True	BOOL	Long	Integer
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 6 is 16-bits on 32-bit platforms. However, in Visual Basic.NET, as well as languages like Visual C++, the default integer size is 32-bits. Also, some languages do not support unsigned integer types. In this case, as with Visual Basic, the signed type should be used instead.

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. In languages like Visual C++, boolean parameters should always be passed as 32-bit signed integers.

String Data

String arguments can also present a problem when calling methods from languages such as Visual C++. All strings, regardless of whether they are assigned to property values or to be passed as arguments, must be specified as BSTRs. A BSTR is essentially a null-terminated Unicode string with the length of the string prepended to it. Each character in the BSTR is represented as a 16-bit value. With languages such as Visual Basic, strings are handled transparently. However, in C++ it is required that those strings be allocated and managed by the application. It is recommended that you use classes like CComBSTR to represent your string values.

If you are unsure of how your language handles BSTRs, we recommend that you review the language's technical reference for information on how to assign string values to the property of a COM object, or when calling a method. If your language supports COM interfaces, it will typically either handle BSTR strings transparently or provide a collection of functions which can be used to create, modify and delete them.

Unicode

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.

Whenever a string is assigned to a property value or passed to a method, that string is in Unicode. If necessary, the control will automatically convert that string to ANSI and it does not require any additional programming on the part of the developer. This is all largely transparent when using the components in high-level languages like Visual Basic. However, in Visual C++ and other languages that deal with COM objects on a lower level, it is important to understand that string values must be passed as BSTRs, which are Unicode strings.

The issue that most commonly confronts developers with regards to how strings are handled by the SocketTools components are with regards to the Read and Write methods. These methods are used to send and receive data over the network, and accept several different types of data. Typically, the data is exchanged as either a string of text characters, or as an array of bytes. Consider the following code:

```
Dim strMessage As String
Dim strBuffer As String
Dim cbBuffer As Long

Do

  cbBuffer = SocketWrench1.Read(strBuffer, 1024)
  If cbBuffer > 0 Then strMessage = strMessage + strBuffer

Loop Until cbBuffer < 1
```

In this case, the program expects to receive data from the server which is textual, and it will be stored in the string ***strMessage***. What happens internally is that the data received from the server is automatically converted from an array of bytes into a string by the control. This is done because the control knows that the ***strBuffer*** argument is typed as a **String**, which means it is Unicode. However, what if the data being returned by the server contains binary data or is already Unicode text? In this case, the data may end up being corrupted because of the conversion performed by the control. To prevent this, the solution is to read the data into an array of bytes rather than a string. For example:

```
Dim byteMessage() As Byte
Dim byteBuffer(1024) As Byte
Dim cbMessage As Long
Dim cbBuffer As Long

Do
  cbBuffer = SocketWrench1.Read(byteBuffer, 1024)

  If cbBuffer > 0 Then
    ReDim Preserve byteMessage(cbMessage + cbBuffer) As Byte

    For nIndex = 0 To cbBuffer - 1
      byteMessage(cbMessage + nIndex) = byteBuffer(nIndex)
    Next
    cbMessage = cbMessage + cbBuffer
  End If

Loop Until cbBuffer < 1
```

In this case, because the data is being read into a byte array, not a string, then no Unicode conversion is performed and the data is returned exactly as it was sent. Note that Visual Basic also supports the ability to explicitly convert between Unicode strings and byte arrays using the **StrConv** function. For more information, refer to the language reference and online help in Visual Basic.

Microsoft Visual C++

Although it is possible to reference the SocketTools ActiveX controls in a Visual C++ project, this is not recommended. The SocketTools Library Edition includes a native API and C++ classes, and these should be used whenever possible. There are some limitations and performance penalties which are introduced when using ActiveX with C++ compared to using the native SocketTools C++ classes.

The SocketTools controls can be used in Visual C++ in several ways, depending on the type of program being developed and the way in which the control will be utilized. Although much of the complexity of COM can be hidden through the use of wrapper classes and smart pointers, development using COM objects is still more complex than simply using the MFC classes with which most Windows developers are familiar.

One of the first things that a C++ developer will encounter when programming with the control is that all of the methods use a data type called a ***variant***. Most developers aren't familiar with what a variant is or how it should be used unless they have experience with COM programming, so this is a frequent point of confusion. The simplest definition is that a variant is a structure which contains type information and a union of intrinsic data types such as characters, integers and so on. The variant essentially serves as a generic data type, and the function being called has the responsibility of using or converting that data as necessary. In addition to the VARIANT structure itself, there are several classes which encapsulate variants, such as **_variant_t**, **COleVariant** and **CComVariant**. These classes make it easier for C++ programmers to use variants, and for the most part allows them to be used just as if the variant was an intrinsic type.

Another data type that may be unfamiliar is the BSTR, which is used for string data. Similar to C strings, the BSTR is a pointer to a null terminated array of characters which make up a string. However, there are some significant differences between the two. First, a BSTR always uses the Unicode character set, even if the program itself does not use Unicode. That means that each character in the BSTR is actually 16 bits, so special care must be taken to not assume that a character in the string is equivalent to a single byte. Second, although BSTR strings are null terminated, they may actually contain embedded nulls. This is because the BSTR also has information about the length of the string, so standard string functions (even the Unicode versions of them) should not be used if there is a chance that the string contains embedded nulls. Part of the Automation API is a collection of functions which manage BSTR strings, such as **SysAllocString** and **SysStringLen**. However, most programmers prefer to use one of the classes which encapsulate BSTRs, such as **_bstr_t** and **CComBSTR**.

In addition to the COM data types, another aspect of using COM objects is that most COM related functions return HRESULT values. The HRESULT is a 32-bit unsigned integer which contains status information about an error or warning returned by a function. Two macros which are commonly used are FAILED and SUCCEEDED which are used to determine whether or not the HRESULT value indicates that the function failed or was successful. All COM object methods and property accessor functions return HRESULT values which must be checked by the caller to ensure that the function was called correctly.

Microsoft Foundation Classes

Although it is possible to reference the SocketTools ActiveX controls in a Visual C++ project, this is not recommended. The SocketTools Library Edition includes a native API and C++ classes with support for MFC, and these should be used whenever possible. There are some limitations and performance penalties which are introduced when using ActiveX with C++ compared to using the native SocketTools C++ classes.

The SocketTools controls can be used with MFC based applications by including the control in the project that is being developed. This is done through the Visual C++ IDE by selecting the menu option **Project | Add to Project | Components and Controls**. This will display a dialog which is used to select the component to add. First select the Registered ActiveX Controls folder, scroll over to the control that you're interested in using, and press the Insert button. A dialog is then displayed which determines the class name and files which will be generated to "wrap" the ActiveX control. A new source file will be added to the project which contains the methods for the wrapper class that was created. A header file will also be created and included in the header file for the dialog class.

To create an instance of the control, the simplest approach is to create a dialog-based application, in which case the control can be selected from the dialog component palette, similar to how controls are placed on forms in Visual Basic. The control is included as a resource and assigned a resource ID.

Then, using the MFC Class Wizard, a member variable for the dialog class is assigned to that instance of the control. This means that a declaration similar to this will be added to the dialog class:

```
  CFtpClient m_ctlFtpClient;
```

In the **DoDataExchange** method, a line will be added which initializes the control when the dialog is created:

```
  DDX_Control(pDX, IDC_FTPCLIENT1, m_ctlFtpClient);
```

Now, any of the control's properties or methods may be accessed through the member variable for the CFtpClient class. For example:

```
COleVariant varServerName(m_strServerName);
COleVariant varServerPort(m_nServerPort);
COleVariant varUserName(m_strUserName);
COleVariant varPassword(m_strPassword);
COleVariant varAccount(m_strAccount);
COleVariant varTimeout(m_nTimeout);
COleVariant varLocalFile(m_strLocalFile);
COleVariant varRemoteFile(m_strRemoteFile);
COleVariant varOptions;
COleVariant varError;

varError = m_ctlFtpClient.Connect(varServerName,
                                  varServerPort,
                                  varUserName,
                                  varPassword,
                                  varAccount,
                                  varTimeout,
                                  varOptions);

varError.ChangeType(VT_I4);

if (V_I4(&varError) != 0)
{
    CString strError;
    strError.Format(_T("Unable to connect to %s\n%s"),
        m_strServerName,
        m_ctlFtpClient.GetLastErrorString());
```

```
    AfxMessageBox(strError, MB_ICONEXCLAMATION, 0);
}
else
{
    CString strMessage;
    LONG nBytes ;
    COleVariant varRestartOffset;
    varError = m_ctlFtpClient.GetFile(varLocalFile,
                                      varRemoteFile,
                                      varRestartOffset);

    varError.ChangeType(VT_I4);

    if (V_I4(&varError) != 0)
    {
        CString strError;
        strError.Format(_T("Unable to download %s\n%s"),
                m_strRemoteFile,
                m_ctlFtpClient.GetLastErrorString());
        AfxMessageBox(strError, MB_ICONEXCLAMATION, 0);
    }
    else
    {
        nBytes = m_ctlFtpClient.GetTransferBytes();
        strMessage.Format(_T("Transferred %ld bytes of %s"),
                                    nBytes, m_strRemoteFile);
        AfxMessageBox(strMessage, MB_ICONINFORMATION, 0);
    }
    m_ctlFtpClient.Disconnect();
}
```

In this example, the arguments are converted to variants by initializing **COleVariant** variables which are then passed to the **Connect** method. There are two important things to note here. First, even though the documentation lists some of the arguments as optional, when using the control this way in C++, you must specify all of them. This is because optional parameters really aren't omitted from the method; they are still passed as variants, but instead of having a value, they are initialized to tell the control that they were not specified. This is accomplished here by passing an empty (uninitialized) **COleVariant**, as with the **varOptions** variable. The second important point is that although the method is documented as returning a long integer, the actual return type is a variant that contains a long integer value. Similar considerations apply to the **GetFile** method. By contrast, **GetTransferBytes** corresponds to the long-valued property **TransferBytes**, and not to a method of the control, so it really does return a long integer.

You'll notice that in this code, there are also some macros being used with the variant types. The first one is used when checking the return value from the method:

```
  varError.ChangeType(VT_I4);

  if (V_I4(&varError) != 0)
  {
      .
      .
      .
  }
```

The **ChangeType** method for the **COleVariant** class changes the type of variant, in this case to a long integer, specified by the value **VT_I4**. What this does is coerce the variant data into a long integer if it already isn't one. If the variant already represents a long integer, then the call to **ChangeType** doesn't have any effect. Next, the **V_I4** macro is used to obtain the actual value from the long integer. Note that it expects a pointer to a variant,

not the variant itself.

Instantiating CWnd Based Controls

To create an instance of the control in an MFC application without using a dialog, add the control to the project using the same method described previously. However, instead of placing the control on a dialog using the resource editor, declare an instance of the class that will be using the control. Then, call the **Create** function to create an instance of the control for that class. For example:

```
CRect rcNull;
BSTR bstrLicKey;
BOOL bCreated;
USES_CONVERSION;

bstrLicKey = SysAllocString(T2OLE(CSTOOLS11_LICENSE_KEY));
bCreated = m_ctlFtpClient.Create(NULL,         // window name
                                 0,            // window style
                                 rcNull,       // window rect
                                 this,         // parent window
                                 IDC_CONTROL,  // control ID
                                 NULL,         // persistent storage
                                 FALSE,        // IStorage
                                 bstrLicKey);  // license key

if (bCreated == FALSE)
{
    AfxMessageBox(_T("Control creation failed"), MB_ICONEXCLAMATION);
    EndDialog(0);
}
```

Because the control is not part of the program's resources as in the previous example, an instance of the control must be explicitly created by calling the **Create** method. Because the File Transfer control is not visible at runtime, most of the window arguments are null. However, it is still required that a parent window be specified; in this case, the ***this*** pointer is used. If the class that is using the control is not derived from CWnd, a hidden window can be created and specified as the parent instead.

Another issue is that to create an instance of the control, the application must pass it a runtime license key. This is a BSTR string which is used by the control to determine if it can be used in an application. If this string is NULL, then the control will only load if the current system has a valid development license. If it is not NULL, then the license key is validated and an instance of the control is created. The license key for SocketTools is defined in the **cstools11.h** header file, found in the Include folder where the product was installed. Note that the key value will be NULL for evaluation versions of the control, which means that the application cannot be redistributed until a license has been purchased.

The **SysAllocString** function is used to create the license key BSTR and this requires that the license key be converted to Unicode. In **afxpriv.h** there are several string conversion macros that are useful for converting between ANSI and Unicode. One is OLE2T which converts a Unicode string to an LPTSTR, and the other is T2OLE which converts an LPTSTR to a Unicode string. The **afxpriv.h** header file is not usually included in MFC applications, so it will need to be added to **StdAfx.h** manually.

The USES_CONVERSION macro is required and must be included in the function prior to using any of the conversion macros. In this case, T2OLE is used to convert the ANSI license key string to Unicode, and then that is passed to **SysAllocString** to create a BSTR. It should be noted that OLE2T and T2OLE allocate memory from the stack to do the conversion, so they should not be used with very large amounts of data.

Importing ActiveX Controls

In Visual C++ 6.0 and later versions, using the **#import** compiler directive is an alternative to adding the control to a project through the IDE. Similar to how header files are included in an application, this directive incorporates information from a type library, automatically creating wrapper classes for its interfaces. These classes use smart pointers which handle things like reference counting automatically, and make actually using the control's interface much simpler.

To use this method of referencing a control, the first thing that needs to be done is to import the control into the module where it will be used. This is done using the **#import** directive which can be placed in an appropriate header file. For example, to import the File Transfer Protocol control, you would use:

```
#import "csftpx11.ocx" no_namespace named_guids
```

The **no_namespace** attribute specifies that the interface classes should not be defined in a namespace. Normally, a namespace is created which is based on the name of the control. The **named_guids** attribute tells the compiler to initialize the GUID variables using the standard naming convention.

The next step is to declare a variable that is used to reference an instance of the control. For example, the following could be included in the definition of a class:

```
IFtpClientPtr m_pIFtpClient;
```

Note that the member variable is declared as type **IFtpClientPtr**, which is a specialization of the smart **pointer _com_ptr_t** template class. If errors are encountered when compiling the application indicating that the compiler cannot instantiate an abstract class (because the class contains pure virtual functions) then most likely the member variable was declared as type **IFtpClient**, which is incorrect. Don't forget the "Ptr" on the end of the name.

To use the control, an instance of the control must be created using the **CreateInstance** function. However, before that can be done, the COM subsystem must be initialized by the application. For MFC based applications, this is accomplished by calling the function **AfxOleInit** which is essentially a wrapper around **CoInitializeEx**. This should be done fairly early in the application, typically in the **InitInstance** function of the CWinApp derived application class. Next, the control's **CreateInstance** member function must be called before it is used:

```
HRESULT hr;

hr = m_pIFtpClient.CreateInstance(CLSID_FtpClient);
if (FAILED(hr))
{
    AfxMessageBox(_T("Control creation failed"), MB_ICONEXCLAMATION);
    return;
}
```

The HRESULT return value should be 0, which indicates that an instance of the control was created successfully. If an error is returned, this typically means that **AfxOleInit** (or **CoInitializeEx**) was not called first, or the control has not been registered on the system.

Unlike the previous examples where the initialization of the control was performed automatically or by calling the **Create** function, this instance of the control should be explicitly initialized by calling the **Initialize** method:

```
_variant_t varLicKey;
_variant_t varError;
USES_CONVERSION;

// Create the runtime license key defined in csrtkey11.h
varLicKey = SysAllocString(T2OLE(CSTOOLS11_LICENSE_KEY));
```

```
// Initialize the control
varError = m_pIFtpClient->Initialize(varLicKey);
if (V_I4(&varError) != 0)
{
    AfxMessageBox(_T("Control initialization failed"), MB_ICONEXCLAMATION);
    return;
}
```

Just as in the previous example using the **Create** method, the runtime license key is created by converting it to Unicode and then calling **SysAllocString** to create a BSTR string. Because the control methods use variants, this key is assigned to a variant. Note that the **_variant_t** type is used, which is a COM support class which encapsulates a variant. The **Initialize** method returns a long integer variant which specifies an error code. A value of zero indicates that the control was successfully initialized, while a non-zero value is an error code.

Once the control has been created and initialized, it can be used in a fashion similar to how the previous examples were written:

```
_variant_t varServerName(m_strServerName);
_variant_t varServerPort(m_nServerPort);
_variant_t varUserName(m_strUserName);
_variant_t varPassword(m_strPassword);
_variant_t varAccount(m_strAccount);
_variant_t varTimeout(m_nTimeout);
_variant_t varLocalFile(m_strLocalFile);
_variant_t varRemoteFile(m_strRemoteFile);
_variant_t varOptions;
_variant_t varError;

varError = m_pIFtpClient->Connect(varServerName,
                                  varServerPort,
                                  varUserName,
                                  varPassword,
                                  varAccount,
                                  varTimeout,
                                  varOptions);

if (V_I4(&varError) != 0)
{
    CString strError;
    USES_CONVERSION;

    strError.Format(_T("Unable to connect to %s\n%s"),
                    m_strServerName,
                    OLE2T(m_pIFtpClient->GetLastErrorString()));

    AfxMessageBox(strError, MB_ICONEXCLAMATION, 0);
}
else
{
    CString strMessage;
    _variant_t varRestartOffset;
    LONG nBytes ;

    varError = m_pIFtpClient->GetFile(varLocalFile,
                                      varRemoteFile,
                                      varRestartOffset);
    if (V_I4(&varError) != 0)
    {
        CString strError;
```

```
        USES_CONVERSION;

        strError.Format(_T("Unable to get %s\n%s"),
                    m_strRemoteFile,
                    OLE2T(m_pIFtpClient->GetLastErrorString()));

        AfxMessageBox(strError, MB_ICONEXCLAMATION, 0);
    }
    else
    {
        nBytes = m_pIFtpClient->TransferBytes;
        strMessage.Format(_T("Transferred %ld bytesof %s"),
                          nBytes, m_strRemoteFile);
        AfxMessageBox(strMessage, MB_ICONINFORMATION, 0);
    }
    m_pIFtpClient->Disconnect();
}
```

There are two significant differences between the previous examples which use the control as a CWnd derived class, and this class which is based on COM smart pointers. The first is that methods are accessed through the **m_pIFtpClient** object as a pointer to the interface, so the `->` operator is used. The second is that the control's properties, such as **TransferBytes**, can be accessed as if they are member variables of the class rather than using accessor functions like **GetTransferBytes**. This is a bit of slight-of-hand being performed by the interface class using the **__declspec**(property) extension. For example, the **TransferBytes** member is declared as:

```
__declspec(property(get=GetTransferBytes)) long TransferBytes;
```

This tells the compiler whenever the **TransferBytes** member is read, it should call the **GetTransferBytes** function to return the value. So, in effect the above code is changed by the compiler into:

```
nBytes = m_pIFtpClient->GetTransferBytes();
```

Either method may be used, so it is generally up to the personal preferences of the developer as to which is used.

Component Object Model API

Another approach that can be used to create an instance of an ActiveX control in your C++ program is to use the COM API directly. Generally speaking this option should only be used if absolutely necessary; it is a more complex process and involves more coding than either using CWnd derived controls or the **#import** directive.

The first step is to create the header file for the interface defined in the control's type library. This will require two tools that are included with Visual C++ and the Microsoft Windows SDK; the COM Object Viewer, and the Microsoft Interface Definition Language (MIDL) compiler. These tools can also be downloaded from Microsoft from their MSDN resources section of the website.

To create the interface definition (IDL) file, start the COM Object Viewer, select the Control folder and then the control you are interested in. For purposes of this example, we'll use the File Transfer Protocol control. Right click on the control and select View Type Information. This will open the ITypeLib Viewer window which contains the interface definition. Select **File | Save As** and save it as **csftpctl.idl** in the project directory. Close the viewer window and exit the COM Object Viewer.

Once the IDL file has been created, open the IDL file in the editor and look for a series of enum typedefs which define the constants for the control:

```
typedef [public]
    _ftpOptionsConstants ftpOptionsConstants;

    typedef enum {
        ftpOptionHttpNocache = 1,
        ftpOptionFtpSecureAuth = 8192
    } _ftpOptionsConstants;
```

These declarations are a side-effect of how the COM Object Viewer generates the IDL file and it needs to be cleaned up a bit so that the MIDL compiler generates the correct header file. Remove the **typedef [public]** section before each typedef enum section in the file. In other words, you would want to remove each section that looks like:

```
typedef [public]
    _ftpOptionsConstants ftpOptionsConstants;
```

The actual enum typedefs can stay in the IDL so that they're included in the header file and can be used by the application. Note that if these extraneous typedefs aren't removed, the MIDL compiler will generate duplicate enums in the header file and will cause compiler errors.

Save the IDL file and then use the MIDL compiler to generate the header file which will be included with your project. From the command line, enter:

```
midl /Oicf /W1 /Zp8 /h csftpctl.h /iid csftpctl_i.c csftpctl.idl
```

This will create three files: **csftpctl.h**, **csftpctl_i.c** and **csftpctl.tlb**. The TLB is the compiled type library and isn't needed for this example. The **csftpctl.h** header file contains the interface definition for the control, and the **csftpctl.c** file is a C source file which defines the GUIDs used by the control. Both of these files should be included in your project, typically in the source module where the control will be used.

Now that the header file for the control interface has been created, the next step is to create an instance of the control. Define a member variable that is a pointer to the interface which looks like this:

```
IFtpClient *m_pIFtpClient;
```

Safe programming practices would also ensure that the pointer is initialized to NULL in the constructor to avoid potential errors when referencing the variable. As with the previous examples, the COM subsystem must be initialized. For MFC based applications, this can be done by calling **AfxOleInit** in the **InitInstance** function for the CWinApp derived application class. For other applications, **CoInitializeEx** should be called as:

```
HRESULT hr = CoInitializeEx(NULL, COINIT_APARTMENTTHREADED);
if (FAILED(hr))
{
    // Unable to initialize COM subsystem
    return;
}
```

Then the following code can be used to create an instance of the control:

```
HRESULT hr;
BSTR bstrLicKey;
IClassFactory2 *pFactory = NULL;
IUnknown *pUnknown = NULL;
USES_CONVERSION;

m_pIFtpClient = NULL;

hr = CoGetClassObject(CLSID_FtpClient,
                      CLSCTX_INPROC_SERVER,
                      NULL,
                      IID_IClassFactory2,
                      (LPVOID *)&pFactory);

if (FAILED(hr))
{
    // Unable to get the class factory interface for the
    // control, probably because it isn't registered
    EndDialog(0);
    return FALSE;

}

// Create the runtime license key defined in csrtkey11.h
bstrLicKey = SysAllocString(T2OLE(CSTOOLS11_LICENSE_KEY));

// Create an instance of the control
hr = pFactory->CreateInstanceLic(NULL, NULL, IID_IUnknown,
                                 bstrLicKey,
                                 (LPVOID *)&pUnknown);
pFactory->Release();

if (FAILED(hr))
{
    // Unable to create an instance of the control using
    // the specified license key
    EndDialog(0);
    return FALSE;
}

hr = pUnknown->QueryInterface(IID_IFtpClient,
                              (LPVOID *)&m_pIFtpClient);

if (FAILED(hr))
{
    // Unable to get the interface to the control
    EndDialog(0);
    return FALSE;
}
```

The **CoGetClassObject** function is used to get an interface pointer to the control's class factory, which actually

does the work of creating an instance of the class. The **CreateInstanceLic** member function passes the runtime license key to the control, and an instance is created if the key is valid. Note that if a NULL value is passed as the license key, then the control will only be created if the system has a development license installed. The interface to the class factory is released and then **QueryInterface** is called on the returned pointer to obtain the interface to the control's properties and methods.

The code to use the interface is similar to the previous examples, however there are several significant differences:

```
COleVariant varServerName(m_strServerName);
COleVariant varServerPort(m_nServerPort);
COleVariant varUserName(m_strUserName);
COleVariant varPassword(m_strPassword);
COleVariant varAccount(m_strAccount);
COleVariant varTimeout(m_nTimeout);
COleVariant varLocalFile(m_strLocalFile);
COleVariant varRemoteFile(m_strRemoteFile);
COleVariant varOptions;
COleVariant varError;
HRESULT hr;

hr = m_pIFtpClient->Connect(varServerName,
                            varServerPort,
                            varUserName,
                            varPassword,
                            varAccount,
                            varTimeout,
                            varOptions,
                            &varError);

if (V_I4(&varError) != 0)
{
    CString strError;
    BSTR bstrError;
    USES_CONVERSION;

    hr = m_pIFtpClient->get_LastErrorString(&bstrError);
    if (FAILED(hr))
        return;

    strError.Format(_T("Unable to connect to %s\n%s"),
                    m_strServerName,
                    OLE2T(bstrError));

    AfxMessageBox(strError, MB_ICONEXCLAMATION, 0);
}
else
{
    CString strMessage;
    COleVariant varRestartOffset;
    LONG nBytes = 0;

    hr = m_pIFtpClient->GetFile(varLocalFile,
                                varRemoteFile,
                                varRestartOffset,
                                &varError);

    if (V_I4(&varError) != 0)
    {
```

```
        CString strError;
        BSTR bstrError;
         USES_CONVERSION;

        hr = m_pIFtpClient->get_LastErrorString(&bstrError);
        if (FAILED(hr))
            return;

        strError.Format(_T("Unable to download %s\n%s"),
                    m_strRemoteFile,
                    OLE2T(bstrError));

        AfxMessageBox(strError, MB_ICONEXCLAMATION, 0);
    }
    else
    {
        hr = m_pIFtpClient->get_TransferBytes(&nBytes);
        if (FAILED(hr))
            return;
        strMessage.Format(_T("Transferred %ld bytes of %s"),
                          nBytes, m_strRemoteFile);
        AfxMessageBox(strMessage, MB_ICONINFORMATION, 0);
    }

    m_pIFtpClient->Disconnect(&varError);


}
```

As with the version of the code using the COM smart pointer, **p_IFtpClient** is a pointer to the interface, which requires that the -> operator be used to access its member functions. Property values are read using accessor functions that are prefixed with "**get_**", while those which set properties are prefixed with "**put_**". For example, to get the value of the **TransferBytes** property, the function name would be **get_TransferBytes**. Methods in the control are called using the same name.

Another difference is that all of the functions return HRESULT values, with the actual property value or return value from the method specified as a function parameter that is passed by reference. This is why the **varError** variable is passed as the last argument to the **Connect** method. If the HRESULT return value is non-zero, this typically will indicate an error. The error may be specific to the control, or it may be a general error coming from the COM subsystem.

Once the application is done using the control, the interface must be released with code like this:

```
if (m_pIFtpClient)
    m_pIFtpClient->Release();
```

Each control that is created has a reference count which is used to keep track of how many times one of its interfaces has been requested. When the reference count drops to zero, the control destroys itself and releases the memory that was allocated. Failing to release the interface will prevent the control from ever being destroyed and will result in a memory leak.

Control Event Handling

In languages like Visual Basic, using the events for a control simply involves adding code for the desired event. Many of the details, such as connecting the control's event interface to the container, are largely invisible to the programmer. However, when using a control in Visual C++, some extra work does need to be done. This section will cover two basic methods; one is specific to CWnd derived controls which are placed in a dialog, the other approach uses a CCmdTarget derived class to handle event notifications.

To create an event handler for a control that has been placed on a dialog form, open the form in the resource editor, right click the control and select Events. This will open a dialog that lists the available events for the control. Selecting one of the events adds the event to the dialog class with a name like **OnProgressFtpClient1**. In the implementation for the dialog class, a section of code will be added that looks like this:

```
BEGIN_EVENTSINK_MAP(CExampleDlg, CDialog)
  //{{AFX_EVENTSINK_MAP(CExampleDlg)
  ON_EVENT(CExampleDlg, IDC_FTPCLIENT1, 4, OnProgressFtpClient1,
           VTS_VARIANT VTS_VARIANT VTS_VARIANT)
  //}}AFX_EVENTSINK_MAP
END_EVENTSINK_MAP()
```

This is the event sink map which is used to map a function in the dialog class to the control's event dispatch interface. The ON_EVENT macro defines the event sink with the dialog class name, the control ID, the dispatch ID for the event, the event handler function and then the parameters that are passed to the event. The three VTS_VARIANT macros specify that the event handler has three VARIANT arguments. All of this code is automatically generated with one ON_EVENT for each control event that was selected. Of the two approaches, this is the simplest but it depends on the fact that the control has been placed on a dialog.

A more general purpose way to implement event handling is to derive a class from the CCmdTarget class which will act as the event sink for the control. First, edit the **StdAfx.h** header file to include **afxctl.h**. Next, create a new class for the project called CEventSink. It should be derived from CCmdTarget with Automation support enabled. However, do not make it creatable by type ID. A dialog may be displayed that it was unable to edit the object definition (ODL) file for the product. Since your project may not have one, this is only a warning and can be ignored.

Open the **EventSink.cpp** implementation file and look towards the end of the file where there is a section that looks something like this:

```
BEGIN_INTERFACE_MAP(CEventSink, CCmdTarget)
    INTERFACE_PART(CEventSink, IID_IEventSink, Dispatch)
END_INTERFACE_MAP()
```

This maps the CEventSink class to the event interface. This needs to be changed so that it is mapped to the control's **IFtpClientEvents** interface. Change the second argument of the INTERFACE_PART macro to the value **DIID__IFtpClientEvents**. This section should now look like:

```
BEGIN_INTERFACE_MAP(CEventSink, CCmdTarget)
    INTERFACE_PART(CEventSink, DIID__IFtpClientEvents, Dispatch)
END_INTERFACE_MAP()
```

Next, decide what event handlers should be implemented for the control. This example will implement all of them, but it isn't necessary if they aren't actually going to be used by the application. The event handlers will be protected member functions of the CEventSink class. They will be defined in **EventSink.h** and implemented in **EventSink.cpp**:

```
void OnCancel();
void OnCommand(VARIANT& varResultCode, VARIANT& varResultString);
void OnError(VARIANT& varError, VARIANT& varDescription);
```

```
void OnProgress(VARIANT& varFileName, VARIANT& varFileSize, VARIANT&
varBytesCopied,
                VARIANT& varPercent);
void OnTimeout();
```

Once the event handler functions have been implemented, they need to be added to the dispatch map. Look for the BEGIN_DISPATCH_MAP section in **EventSink.cpp** and add the definitions for the events:

```
DISP_FUNCTION_ID(CEventSink,"OnCancel",1,OnCancel,VT_EMPTY,VTS_NONE)
DISP_FUNCTION_ID(CEventSink,"OnCommand",2,OnCommand,
                 VT_EMPTY,VTS_VARIANT VTS_VARIANT)
DISP_FUNCTION_ID(CEventSink,"OnError",3,OnError,
                 VT_EMPTY,VTS_VARIANT VTS_VARIANT)
DISP_FUNCTION_ID(CEventSink,"OnProgress",4,OnProgress,
                 VT_EMPTY,VTS_VARIANT VTS_VARIANT VTS_VARIANT VTS_VARIANT)
DISP_FUNCTION_ID(CEventSink,"OnTimeout",5,OnTimeout,VT_EMPTY,VTS_NONE)
```

These declarations are similar to those used with the ON_EVENT macros in the previous example. The VT_EMPTY type specifies that the event handler does not return a value. VTS_VARIANT specifies a VARIANT argument. VTS_NONE specifies that the event doesn't have any arguments.

With the event handlers implemented, the next step is to connect them to the control. Include the **EventSink.h** header file in the module where the control is being used and create two new member variables for the class:

```
DWORD m_dwEventSink;
CEventSink* m_pEventSink;
```

Next, an instance of the CEventSink class needs to be created and then the sink dispatch interface needs to be connected to the control using the **AfxConnectionAdvise** function:

```
// Create an instance of the event sink class
m_pEventSink = new CEventSink();

// Get a pointer to the sink IDispatch interface
LPUNKNOWN pUnknownSink = m_pEventSink->GetIDispatch(FALSE);

// Connect the event source to the sink
AfxConnectionAdvise(m_pIFtpClient,
                    DIID__IFtpClientEvents,
                    pUnknownSink,
                    FALSE,
                    &m_dwEventSink);
```

The last thing that needs to be done is to disconnect the event sink from the control when it is no longer needed. This is done by calling **AfxConnectionUnadvise**, typically right before an instance of the control is deleted:

```
if (m_pEventSink)
{
    LPUNKNOWN pUnknownSink = m_pEventSink->GetIDispatch(FALSE);

    AfxConnectionUnadvise(m_pIFtpClient,
                          DIID__IFtpClientEvents,
                          pUnknownSink,
                          FALSE,
                          m_dwEventSink);

    delete m_pEventSink;
    m_pEventSink = NULL;
    m_dwEventSink = 0;
}
```

With this code, the application is now wired to receive event notifications from the control. Keep in mind that because the instance of the CEventSink class was created on the heap, failure to destroy the sink will cause a memory leak in the application.

Microsoft Visual Basic .NET

Although it is possible to reference the SocketTools ActiveX controls in a Visual Basic.NET project, this is not recommended. The SocketTools .NET Edition includes assemblies which are designed specifically for .NET languages like Visual Basic, and those .NET components should be used whenever possible. There are a number of significant limitations and performance penalties which are introduced when using ActiveX with modern .NET applications.

The easiest way to create an instance of the control is to add it to your toolbox and place it on a form. Simply select the Toolbox, and right-click on it to bring up a context menu, and then select **Add/Remove Items**. This will display the Customize Toolbox dialog. Select the COM Components tab, and scroll down to the control or controls that you wish to add, check them and click Ok. Once the components have been added to your toolbox, you can drag and drop them on your form, which will add them to your project.

Property Names

In some cases, Visual Basic.NET will rename certain properties to avoid naming conflicts within the component class. The properties function in the same way, but the prefix 'Ctl' is added to the name. The two most common properties that this is done for are the **Handle** and **State** properties, renamed to **CtlHandle** and **CtlState** respectively. For the File Transfer Protocol control, the **System** property is also renamed to **CtlSystem**.

Event Handlers

SocketTools uses events to notify the application when some change in status occurs, such as when data is available to be read or a command has been executed. To create an event handler, it is the same as how you would do it in earlier versions of Visual Basic: select the control from the drop-down listbox on the left side of the code window, and then select the event in the listbox on the right side. A new event handler will be added to your code. For example, if you choose the **OnCommand** event then code like this will be added:

```
Private Sub AxFtpClient1_OnCommand( _
    ByVal sender As Object, _
    ByVal e As AxFtpClientCtl._IFtpClientEvents_OnCommandEvent _
    ) Handles AxFtpClient1.OnCommand

End Sub
```

If you are familiar with Visual Basic 6.0 but new to Visual Basic .NET, the first thing that you'll notice is that the arguments are not passed individually to the event. Instead, they are passed in the argument 'e', which is a class that encapsulates all of the event arguments. The technical reference shows that there are two arguments for the **OnCommand** event, **ResultCode** and **ResultString.** To access their values, you would write code like this:

```
Private Sub AxFtpClient1_OnCommand( _
    ByVal sender As Object, _
    ByVal e As AxFtpClientCtl._IFtpClientEvents_OnCommandEvent _
    ) Handles AxFtpClient1.OnCommand

    Dim nResult As Integer = e.resultCode
    Dim strResult As String = e.resultString

    Console.WriteLine("{0}: {1}", nResult, strResult)
End Sub
```

Note that the arguments passed to the event handler are variants, just as they are with methods. In this case, the numeric result code and result string are written out to the console for each command that is executed on the server.

Exception Handlers

When setting a property, it is possible that the control will generate an exception as a result of an error. If these exceptions are not handled in your application, it will cause the program to halt and display a dialog box. To handle an exception, use the **Try..Catch** statement, such as:

```
Try
    axFtpClient1.HostAddress = "abcd"
Catch comError As System.Runtime.InteropServices.COMException
    Console.WriteLine(comError.Message)
    Exit Sub
End Try
```

In this example, the **HostAddress** property is being set to an illegal value (the **HostAddress** property only accepts IP addresses in dot notation, such as "192.168.0.1"). As a result, an exception is thrown and the Catch section of code is executed. The ***ex*** argument contains information about the exception that has occurred, including an error number and a description of the error. In this case, that description is written to the console and the subroutine is exited.

In addition to exceptions generated when properties are set to invalid values, it is also possible to have methods generate exceptions instead of returning error codes. To do this, set the **ThrowError** property to True. It allows you to write code like this:

```
Try
    axFtpClient1.ThrowError = True
    axFtpClient1.Connect()
    axFtpClient1.ChangeDirectory(strDirName)
    axFtpClient1.ThrowError = False
Catch comError As System.Runtime.InteropServices.COMException
    axFtpClient1.ThrowError = False
    Console.WriteLine("Error {0}: {1}", Hex(comError.ErrorCode), comError.Message)
    If axFtpClient1.IsConnected Then axFtpClient1.Disconnect()
    Exit Sub
End Try
```

In this example, the **ThrowError** property is set to True, and then the **Connect** and **ChangeDirectory** methods are called. If either of these methods fail, an exception will be thrown and the code in the Catch section will be executed. In this case, the error code and description is written to the console, the client is disconnected and the subroutine returns. By setting the **ThrowError** property and writing your code to use exception handling, it enables you to easily group function calls together and handle any potential errors, rather than individually checking the return value for each method.

Microsoft Visual C#

Although it is possible to reference the SocketTools ActiveX controls in a Visual C# project, this is not recommended. The SocketTools .NET Edition includes assemblies which are designed specifically for .NET languages like C#, and those .NET components should be used whenever possible. There are a number of significant limitations and performance penalties which are introduced when using ActiveX with modern .NET applications.

The easiest way to create an instance of the control is to add it to your toolbox and place it on a form. Simply select the Toolbox, and right-click on it to bring up a context menu, and then select **Add/Remove Items**. This will display the Customize Toolbox dialog. Select the COM Components tab, and scroll down to the control or controls that you wish to add, check them and click OK. Once the components have been added to your toolbox, you can drag and drop them on your form, which will add them to your project.

Another approach is to add the control to the project as a reference, rather than placing the control on a form. In the Solution Explorer, right click on References and select Add Reference. Then select the component from the list of COM objects and click OK. Once the control has been added to the project as a reference, an instance of the control can be created using the **new** operator.

Variants and Objects

The SocketTools component interface is essentially the same in C# as it is with other languages, however there are a few differences. In C#, variant data types are represented by the **object** type. This is important because all of the component methods and event arguments are typed as variants. This can lead to unexpected errors when developing your application. For example, consider this code:

```
int nError;
nError = axFtpClient1.Connect();
```

Because the Connect method returns a value which indicates success by returning a value of zero, or an error code, this code looks correct. However, if you would try to compile it, you would get an error message indicating that the compiler cannot implicitly convert type 'object' to type 'int'. This is because all of the methods in the SocketTools components return the value as a variant type, which in C# is represented as an object. To resolve this, all you need to do is simply cast the return value to an int type, such as:

```
int nError;
nError = (int)axFtpClient1.Connect();
```

Another common issue is that some of the methods in the SocketTools components expect arguments to be passed by reference, and the data is returned in the specified variable. An example of this is the **GetDirectory** method in the File Transfer Protocol control. The method expects a single argument, passed by reference. When the method returns, that argument will contain a string that specifies the current working directory. Because the method expects a variant type, you simply can't pass a string variable to the method. Instead, you need to use code like this:

```
object varDirectory = "";
int nError;

nError = (int)axFtpClient1.GetDirectory(ref varDirectory);
if (nError == 0)
{
    string strDirectory = varDirectory.ToString();
}
```

The *varDirectory* object is initialized to an empty string, and the ref keyword tells C# to pass the object by reference to the method. If the method returns a value of zero, which indicates success, then *varDirectory* contains the current working directory and that is assigned to a string variable. Note that this approach will work with string parameters, but will not work correctly with other types such as byte arrays.

Optional Arguments

Many of the methods in the SocketTools components use optional arguments which may be omitted if the caller wishes to use a default value. Unlike Visual Basic, C# requires that all arguments be specified. To omit an optional argument, use the special value **Type.Missing** as a placeholder. For example, the **GetFile** method in the FTP control expects four arguments, with the last two arguments being optional. The method could be called as follows:

```
nError = (int)axFtpClient1.GetFile(strLocalFile, strRemoteFile, Type.Missing,
Type.Missing);
```

In this example, the **GetFile** method will use default values for the last two arguments. Note that this should only be used with those arguments which the documentation specifies as optional. If the argument is required and **Type.Missing** as passed as the value, a runtime exception will be thrown.

Property Names

In some cases, Visual C# will rename certain properties to avoid naming conflicts within the component class. The properties function in the same way, but the prefix 'Ctl' is added to the name. The two most common properties that this is done for are the **Handle** and **State** properties, renamed to **CtlHandle** and **CtlState** respectively. For the File Transfer Protocol control, the **System** property is also renamed to **CtlSystem**.

Note that these changes to the property names only occurs when the control is placed on a form, not when the control is added to the project as a reference.

Event Handlers

SocketTools uses events to notify the application when some change in status occurs, such as when data is available to be read or a command has been executed. To create an event handler, simply select the control, click on the Events button in the property browser and then double-click on the event. A new event handler will be added to your code. For example, if you choose the **OnCommand** event then code like this will be added:

```
private void axFtpClient1_OnCommand(object sender,
AxFtpClientCtl._IFtpClientEvents_OnCommandEvent e)
{

}
```

If you are familiar with Visual Basic but new to C#, the first thing that you'll notice is that the arguments are not passed individually to the event. Instead, they are passed in the argument 'e', which is a class that encapsulates all of the event arguments. The technical reference shows that there are two arguments for the **OnCommand** event, ***ResultCode*** and ***ResultString***. To access their values, you would write code like this:

```
private void axFtpClient1_OnCommand(object sender,
AxFtpClientCtl._IFtpClientEvents_OnCommandEvent e)
{
    int nResult = (int)e.resultCode;
    string strResult = e.resultString.ToString();

    Console.WriteLine("{0}: {1}", nResult, strResult);
}
```

Note that the arguments passed to the event handler are variants, just as they are with methods. This means that in C# they are treated as the object type, and should be cast or explicitly converted to other intrinsic types.

Exception Handlers

When setting a property, it is possible that the control will generate an exception as a result of an error. If these exceptions are not handled in your application, it will cause the program to halt and display a dialog box. To handle an exception, use the try..catch statement, such as:

```
try
{
    axFtpClient1.HostAddress = "abcd";
}
catch (System.Runtime.InteropServices.COMException ex)
{
    Console.WriteLine(ex.Message.ToString());
    return;
}
```

In this example, the **HostAddress** property is being set to an illegal value (the **HostAddress** property only accepts IP addresses in dot notation, such as "192.168.0.1"). As a result, an exception is thrown and the ***catch*** section of code is executed. The ***ex*** argument contains information about the exception that has occurred, including an error number and a description of the error. In this case, that description is written to the console and the function is exited.

In addition to exceptions generated when properties are set to invalid values, it is also possible to have methods generate exceptions instead of returning error codes. To do this, set the **ThrowError** property to true. It allows you to write code like this:

```
try
{
    axFtpClient1.ThrowError = true;
    axFtpClient1.Connect();
    axFtpClient1.ChangeDirectory(strDirName);
    axFtpClient1.ThrowError = false;
}
catch (System.Runtime.InteropServices.COMException ex)
{
    axFtpClient1.ThrowError = false;
    Console.WriteLine("Error {0}: {1}", ex.ErrorCode, ex.Message.ToString());

    if (axFtpClient1.IsConnected)
    {
        axFtpClient1.Disconnect();
    }
    return;
}
```

In this example, the **ThrowError** property is set to true, and then the **Connect** and **ChangeDirectory** methods are called. If either of these methods fail, an exception will be thrown and the code in the ***catch*** section will be executed. In this case, the error code and description is written to the console, the client is disconnected and the function returns. By setting the **ThrowError** property and writing your code to use exception handling, it enables you to easily group function calls together and handle any potential errors, rather than individually checking the return value for each method.

SocketTools Control Overview

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

Domain Name Service Control

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

File Encoding Control

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

File Transfer Protocol Control

The File Transfer Protocol (FTP) control provides methods for uploading and downloading files from a server, as well as a variety of remote file management methods. In addition to file transfers, an application can create, rename and delete files and directories, list files and search for files using wildcards. The control provides both high level methods, such as the ability to transfer multiple files in a single method call, as well as access to lower level remote file I/O methods.

Hypertext Transfer Protocol Control

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

Internet Control Message Protocol Control

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

Internet Message Access Protocol Control

The Internet Message Access Protocol (IMAP) is an application protocol which is used to access a user's email messages which are stored on a mail server. However, unlike the Post Office Protocol (POP) where messages are downloaded and processed on the local system, the messages on an IMAP

server are retained on the server and processed remotely. This is ideal for users who need access to a centralized store of messages or have limited bandwidth. For example, traveling salesmen who have notebook computers or mobile users on a wireless network would be ideal candidates for using IMAP. The SocketTools IMAP control implements the current standard for this protocol, and provides methods to retrieve messages, or just certain parts of a message, create and manage mailboxes, search for specific messages based on certain criteria and so on. The interface is designed as a superset of the Post Office Protocol interface, so developers who are used to working with the POP3 control will find the IMAP control very easy to integrate into an existing application.

Internet Server Control

The Internet Server 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.

Mail Message Control

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

Network News Transfer Protocol Control

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

News Feed Control

The News Feed control enables an application to download and process a syndicated news feed in in standard RSS format. News feeds can be accessed remotely from a web server, or locally as an XML formatted text file. The source of the feed is determined by the URI scheme that is specified. If the http or https scheme is specified, then the feed is retrieved from a web server. If the file scheme is used, the feed is considered to be local and is accessed from the disk or local network. The News Feed control provides an interface that enables you to open a feed by URL and iterate through each of the items in the feed or search for a specific feed item. The control also provides a method that can be used to parse a string that contains XML data in RSS format, where the feed may have been retrieved from other sources such as a database.

Post Office Protocol Control

The Post Office Protocol (POP) control provides access to a user's new email messages on a mail

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

Remote Access Services Control

The Remote Access Services (RAS) control enables an application to connect to an Internet Service Provider (ISP) using a standard Dial-Up Networking connection. Using this control, the application can discover what dial-up devices are available, what dial-up networking entries, known as "connectoids", are available on the local system and allows the program to manage those connections. Existing connections can be monitored, new connections created and a single control can be used to manage multiple dial-up connections if the system has more than one modem. While Windows can be configured to simply autodial a service provider whenever a network connection is needed, this component gives your application complete control over the process of connecting to a service provider, monitoring that connection and then terminating that connection if needed.

Remote Command Protocol Control

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

Secure Shell Protocol Control

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

Simple Mail Transfer Protocol Control

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

SocketWrench Control

The SocketWrench control provides a higher-level interface to the Windows Sockets API, designed to be suitable for programming languages other than C and C++. In addition, SocketWrench supports secure communications using the Transport Layer Security (TLS) protocol.

Telnet Protocol Control

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

Terminal Emulation Control

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

Text Message Control

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

Time Protocol Control

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

Web Location Control

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

Web Storage Control

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

Whois Protocol Control

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

Domain Name Service

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

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.

Reset
Reset the internal state of the control and re-initialize the component to use the default nameserver configuration for the local host. This can be useful if your application wishes to discard any settings made by a user and return to using the local system configuration.

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.

Host Tables

When resolving a host name or IP address, the library will first search the local system's host table, a file that is used to map host names to addresses. On Windows 95/98 and Windows Me, if the file exists it is usually found in C:\Windows\hosts. On Windows NT and later versions, it is found in C:\Windows\system32\drivers\etc\hosts. Note that the file does not have an extension.

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

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
A property which returns the IP address of the host name specified in the **HostName** property. Setting this property to an IP address will cause the control to perform a reverse DNS lookup to attempt to determine the name of the host that was assigned that address. If successful, the host name for the specified IP address can be determined by reading the value of the **HostName** property.

HostName
A property which returns the name of the host associated with the IP address specified in the **HostAddress** property. Setting this property to a host name will cause the control to perform a DNS lookup to determine the IP address of that host. If successful, the IP address for the host can be

determined by reading the value of the **HostAddress** property.

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

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

Mail Exchange Records

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

MailExchange
A property array which returns the mail exchanges for the domain specified in the **HostName** property. This is a zero-based array, with the maximum number of entries returned by the MailExchanges property

Advanced Properties

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

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

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

NameServer
A property array which can be used to return the current nameservers that are configured for the local host, or the values can be changed to specify new nameservers. The maximum number of nameservers that can be configured for each instance of the control is four.

File Encoding

A common requirement for applications which use Internet protocols is the need to encode binary files, as well as compress data to reduce the bandwidth and time required to send or receive the data. Encoding a binary file converts the contents of the file into printable characters which can be safely transferred over the Internet using protocols that only support a subset of 7-bit ASCII characters. This is commonly a restriction for email, since many mail servers still are not capable of correctly processing messages which contain control characters, 8-bit data or multi-byte character sequences found in International text. To address this problem, the sender encodes and sends the data as part of a message; the recipient then extracts and decodes the data, with the end result being the same as the original, without any potential corruption by the mail servers which store and/or forward the message. The File Encoding control supports several encoding and decoding methods, including standard base64 encoding, quoted-printable encoding and uuencoding. For applications which access USENET newsgroup, the control also supports the yEnc encoding method, which has become popular for attaching binary files to a message.

In addition to encoding and decoding files, the File Encoding control also can be used to compress files, reducing their overall size. Two compression algorithms are supported, the standard deflate algorithm which is commonly used in Zip files, and an algorithm based on the Burrows-Wheeler Transform (BWT) which can offer improved compression over the deflate algorithm for some types of files. The developer has control over the type of compression performed, as well as details such as the level of compression which determines how much memory and CPU time is allocated to compress the data.

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

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

Encoding Types

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

Base64

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

Quoted-Printable

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

Uucode

One of the original encoding methods used for email, it gets its name from two UNIX command-line utilities called uuencode and uudecode, which were used to encode and decode files. Like Base64, uuencoding converts three bytes of data into four six-bit numbers, and then a value of 32 is added to

ensure that it is printable. Uuencoding also adds some additional characters which are used to ensure the integrity of the encoded data. This encoding method is still used when posting files to USENET newsgroups, but has largely been replaced by Base64 when attaching files to email messages.

yEnc

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

Data Encoding

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

EncodeFile

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

DecodeFile

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

Data Compression

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

CompressFile

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

ExpandFile

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

Note that there are advanced options for compressing files, such as the ability to specify the compression type and level. Please refer to the Technical Reference for more information.

File Transfer Protocol

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

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 server, using either a host name or IP address. The method has several options related to security as well as the general operation of the control. If the local system is behind a firewall or a route which uses Network Address Translation (NAT), it is recommended that you make sure the **Passive** property is set to true before establishing the connection. This will ensure that your application only uses outbound connections to the server.

Login

Authenticate the client session, providing the server with a user name, password and optionally an account name. It is also possible to use an anonymous (unauthenticated) session by providing empty strings as the username and password. If the **UserName** and **Password** properties are set prior to connecting, the user will automatically be logged in. This method is only necessary if the application needs to access the server using different user accounts during the same session.

Disconnect

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

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.

File Transfers

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

GetData

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

GetFile

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

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

PutData
This method creates a file on the server containing the data that you provide. This can be useful if your application wants to upload dynamically created content without having to create a temporary file on the local system. The data may be specified either as a string, or as the contents of a byte array.

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

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

File Management

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

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

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

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

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

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

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

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

SetFilePermissions

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

Directory Management

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

ChangeDirectory

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

MakeDirectory

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

OpenDirectory

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

ReadDirectory

Return information about the next file in the directory that has been opened. This method is called repeatedly until it indicates that all of the files have been returned.

RemoveDirectory

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

Hypertext Transfer Protocol

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

File Transfers

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

Script Execution

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

Uniform Resource Locators

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

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

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

Understanding how a URL is constructed will help in understanding how the different methods in the control work together. For example, the server name and port number portion of the URL are the values passed to the Connect method to establish the connection. The user name and password values are assigned to the **UserName** and **Password** properties to authenticate the client session. And the resource name is passed to the **GetData** or **GetFile** methods to transfer it to the local system.

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
Establish a connection to the server. Once the connection has been established, the other methods in the control may be used to access the resources on the server.

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

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.

File Transfers

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

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

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

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

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

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

File Management

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

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

GetFileTime

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

DeleteFile

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

Command

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

Script Execution

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

GetData

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

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

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

PostData

An alternative method of providing information to a script is to post data to the script. Instead of the data being part of the resource name itself, posted data is sent separately and is provided as input to the script. This is the same method that is typically used when a user clicks the Submit button on a web-based form. This method requires the name of the script and the address of a buffer that contains the data that will be posted. The resulting output from the script is returned to the caller in the same way that the GetData method works.

Internet Control Message Protocol

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

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.

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.

Ping and TraceRoute

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

Echo

This is the simplest method you can use to send ICMP echo datagrams. Specify the server, the size of the ICMP datagram you want to send and the number of times you want to send it. The method will return if the operation was successful along with information such as the average number of milliseconds it took for the datagram to be returned by the server.

TraceRoute

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

OnTrace

This event is generated when the **TraceRoute** method is called. The event will fire for each intermediate host in the route from the local system and the server.

Remote Access Services

The Remote Access Services (RAS) control enables an application to connect to an Internet Service Provider (ISP) using a standard Dial-Up Networking connection. Using this control, the application can discover what dial-up devices are available, what dial-up networking entries, known as "connectoids", are available on the local system and allows the program to manage those connections. Existing connections can be monitored, new connections created and a single control can be used to manage multiple dial-up connections if the system has more than one modem. While Windows can be configured to simply autodial a service provider whenever a network connection is needed, this component gives your application complete control over the process of connecting to a service provider, monitoring that connection and then terminating that connection if needed.

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

Initialize

Initialize the control and load the Remote Access Services libraries 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

Establish a connection to the dial-up networking server. Once the connection has been established, the control will authenticate the session and the local system will have a network connection to the service provider.

Disconnect

Disconnect from the server and release any resources that have been allocated for the dial-up networking session. After this method is called, the session is no longer valid.

Uninitialize

Unload the Remote Access Services libraries 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.

Connection Properties

The properties of the control are used to set or return information about the current dial-up networking connection. To load a dial-up networking connection, called a connectoid or phonebook entry, use the **LoadEntry** method. There are a large number of properties, however the most significant of those properties are as follows:

DeviceName

This property specifies the name of the device that is used to establish the dial-up networking connection. In most cases this is the name of an analog modem using a serial communications port, connected to a standard telephone line. If your application needs to enumerate the available dial-up networking devices, refer to the **DeviceCount**, **DeviceEntry** and **DeviceType** properties.

DynamicAddress

This property determines if the dial-up networking connection uses a dynamically assigned IP address returned by the server, or a specific IP address configured on the local host. In most cases, this property should be set to True, unless otherwise specified by your service provider.

DynamicNameserver

This property determines if the dial-up networking connection uses dynamically assigned nameservers, used to resolve domain names into IP addresses. In most cases, this property should be

set to True. If your service provider requires that you explicitly specify the nameservers to use, then set this property to False and set the **NameServer** property array to the address of the nameserver(s) to use.

InternetAddress
This property returns the IP address assigned to the current dial-up networking session, if a connection has been established. It can also be used to explicitly specify an IP address if the **DynamicAddress** property is set to False.

NameServer
This is a property array which specifies the IP addresses of the nameservers that are to be used for the current dial-up networking session. If a connection has been established, this property array will return the addresses of those nameservers that have been assigned to you. If the DynamicNameserver property is set to False, this property array can also be used to explicitly specify the nameservers to be used by the dial-up networking connection.

Password
This property specifies the password used to authenticate the dial-up networking connection.

PhoneEntry
This property specifies the name of the connectoid for the current dial-up networking connection. If no connection is active and no connectoid has been loaded, then this property will return an empty string.

PhoneNumber
This property specifies the telephone number for the dial-up networking connection. You should also check the value of the **CountryCode** property, which will tell your application if area code dialing rules are being used. If the **CountryCode** property is set to zero, then no area code dialing rules are in effect and the telephone number is dialed as-is. Otherwise you should check the value of the **AreaCode** property if you need to determine the area code being used for the connection.

UserName
This property specifies the username used to authenticate the dial-up networking connection.

Managing Connectoids
A connectoid contains the information needed to establish a connection, and is represented as the icon in the Network Connections for the local system. Connectoids are referenced by name and typically are named after the service provider, such as "EarthLink" or "Verizon". In addition to simply connecting to a dial-up networking server, the control also enables your application to create, edit and delete these connectoids. Note that in the control documentation, connectoids are also referred to as "entry names" or "phonebook entries". The connectoids are stored as entries in a database files called "phonebooks" and in most cases, we recommend that you simply use the default phonebook.

CreateEntry
This method displays a dialog box that allows the user to specify the information needed to create a new connectoid. This is similar to the dialog that is displayed whenever the user chooses to create a new Dial-Up Networking connection. Note that if you want to create a connectoid without showing a dialog to the user, use the **SaveEntry** method instead.

DeleteEntry
This method deletes an existing dial-up networking connection. Exercise caution when using this method; once a connectoid has been deleted, there is no way to recover it.

LoadEntry
This method loads an existing connectoid, and updates the control's properties to reflect the connectoid's settings. Changing one or more of those properties and then calling the **SaveEntry**

method is how you can modify an existing connectoid.

RenameEntry

This method renames an existing connectoid.

SaveEntry

This method modifies or creates a new connectoid based on the current properties of the control. If the connectoid already exists, it is modified, otherwise a new connectoid is created. Unlike the **CreateEntry** method, this method will not display any dialogs, so it is the responsibility of the application to provide a user interface if needed.

Internet Message Access Protocol

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

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
Establish a connection to the IMAP server. Once the connection has been established, the other methods in the control may be used to access the messages on the server.

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

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.

Managing Mailboxes

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

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

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

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

ExamineMailbox

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

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

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

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

Managing Messages

There are methods in the IMAP control for managing messages which enables the application to create, delete and move messages. To use these methods, a mailbox must be selected, either by setting the **MailboxName** property or calling the **SelectMailbox** method. Methods which modify the mailbox require that it be opened in read-write mode. Messages are identified by a number, starting with one for the first message in the mailbox.

CopyMessage
Copy a message to a specific mailbox.

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

UndeleteMessage
Remove the deletion flag from the specified message.

Viewing Messages

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

GetHeader
This method returns the value for a specified header field in the message. Using this method, it is not necessary to download and parse the message header.

GetHeaders
This method retrieves the complete headers for the specified message and stores it in a string or byte array provided by the caller.

GetMessage
This method retrieves the specified message and stores it in a string or byte array provided by the caller; you can specify the type of message data that you want, a specific part of a multipart message and the amount of data that you want. For example, it is possible to request that only the first 1500

bytes of the body of the 3rd part of a multipart message should be returned.

OpenMessage
This method is a lower level method which opens a message for reading from the server. The application would then call Read to read the contents of the message, followed by **CloseMessage** when all the message data has been read. Also see the **GetMessage** method, which will return the contents of a message into a string or byte array.

Downloading Messages

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

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

Internet Server

The Internet Server control provides an interface which is similar to the SocketWrench control, but is specifically designed to simplify the development of a server application. The control provides a collection of methods which can be used to easily create an event-driven server application. The server runs on a separate thread in the background, automatically managing the individual client sessions as servers connect and disconnect from the server. Events are used to notify the application when the client establishes a connection with the server, sends data to the server or disconnects. Methods such as Read and Write are used to exchange data with the clients.

It is important to note that although the server is multithreaded, the ActiveX control specification requires that event notifications be marshaled across threads. This means that the event handler code that is written executes in the context of the thread that created the control, typically the main UI thread. For languages such as Visual Basic 6.0, the Internet Server control can be used to easily create a server which is designed for a limited number of active client connections. However, for higher volume servers it is recommended that you use a language that fully supports multithreading.

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.

Start

This method starts the server, creating the background thread and listening for incoming client connections on the specified port number. You can specify the local address, port number, backlog queue size and the maximum number of clients that can establish a connection with the server.

Restart

This method will terminate all active client connections, close the listening socket and re-create a new listening socket bound to the same address and port number.

Suspend

This method instructs the server to temporarily suspend accepting new client connections. Existing connections are unaffected, and any incoming client connections are queued until the server is resumed. It is not recommended that you leave a server in a suspended state for an extended period of time. Once the connection backlog queue has filled, any subsequent client connections will be automatically rejected.

Resume

This function instructs the server to resume accepting client connections after it was suspended. Any pending client connections are accepted after the server has resumed normal operation.

Throttle

This method is used to control the maximum number of clients that may connect to the server, the maximum number of clients that can connect from a single IP address and the rate at which the server will accept client connections. By default, there are no limits on the number of active client sessions and connections are accepted immediately. This method can be useful in preventing denial-of-service attacks where the the attacker attempts to flood the server with connection attempts.

Stop

This method will terminate all active client connections, close the listening socket and terminate the

background thread that manages the server. Any incoming client connections will be refused, and all resources allocated for the server will be released.

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 the client establishes a connection with the server, 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

This method reads data from the client and copy it to the string buffer or byte array provided by the caller. If the client closes its 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 read. This method should always be used when reading binary data from the client into a byte array.

ReadLine

Read a line of text from the client, up to an end-of-line character sequence or when the client closes the connection. This method is useful when the client and server are exchanging textual data, as is common with most command/response application protocols.

Write

This method sends data to the client. If the method succeeds, the return value is the number of bytes actually written. This method should always be used when sending binary data to the client.

WriteLine

Write a line of text to the socket, terminating it with an end-of-line character sequence. This method is useful when the client and server are exchanging textual data, as is common with most command/response application protocols.

Broadcast

Broadcasts data to each of the clients that are connected to the server. This can be useful when the application needs to send the same data to each active client session, such as broadcasting a shutdown message when the server is about to be terminated.

IsReadable

This property is used to determine if there is data available to be read from the client. If the property returns a value of True, the Read method will return without causing the application to block. If the property returns False, there is no data available to read from the current client session.

IsWritable

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

Local Host Information

Several properties are provided to return information about the local host, including its fully qualified domain name and the IP addresses that are configured on the system.

ServerName

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.

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).

AdapterAddress

This 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.

Mail Message

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

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

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

Initialize
Initialize the control 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.

ComposeMessage
Compose a new message using the specified header field values and content. Using this method, you can create a message with the From, To, Cc and Subject headers already defined, along with any text for the message. You can also optionally provide both plain and styled HTML text versions of the message and the method will automatically create a multipart message.

ClearMessage
Releases the memory allocated for the current message, including any file attachments, and creates a new, empty message.

Uninitialize
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.

Message Headers

Each message has one or more headers fields which provide information about the contents of the message. For example, the "From" header field specifies the email address of the person who sent the message. There are a fairly large number of header fields defined by the MIME standard, and applications can also create their own custom headers if they wish. The control gives the application complete access to the header fields in a message. Headers can be examined, modified, created or removed from the message as needed.

GetHeader
This method copies the value of a header field into a string buffer that you provide. To return the value of the common header fields such as "From", "To" and "Subject", you should specify a message part of zero by setting the **MessagePart** property.

GetFirstHeader
This method returns the value of the first header defined in the current message part, copying it into the string buffer that you provide. This is used in conjunction with the **GetNextHeader** method to enumerate all of the headers that have been defined.

GetNextHeader
This method returns the value of the next header defined in the current message part. It should be called in a loop until it returns a value of zero (False) which indicates that the last message header has been returned.

SetHeader
Set a message header field to the specified value in the current message part. If the value is an empty string, the message header will be deleted from the message.

DeleteHeader
Delete the specified message header from the current message part.

Message Contents

The content or body of a message contains the text that is to be read or processed by the recipient. It may be a simple, plain text message or it may be more complex, such as a combination of plain and styled HTML text or the data for a file attachment. The control provides complete access to the contents of the message, enabling the application to modify, extract, replace or delete specific sections of the message.

Message
This property returns the current message, including the headers and all message parts, as a string. Setting this property will cause the current message to be cleared and replaced by the new value. The string contents must follow the standard specifications for a message. If the property is set to an empty string, the current message is cleared.

Text
This property returns the body of the current message part. Setting this property replaces the entire message body with the new text.

SelLength, SelStart, SelText
The **SelText** property returns the selected message body text as specified by the **SelStart** and **SelLength** properties. Setting this property replaces text in the message body starting at the byte offset specified by the **SelStart** property.

Multipart Messages

Most typical messages contain a single part, which consists of the message headers followed by the contents of the message. However, when files are attached to a message or alternative content types such as HTML are used, a more complex multipart message is required. With a multipart message, the contents of the message are split into logical sections with each section containing a specific part of the message. For example, when a file is attached to a message, one part of the message contains the text to be read by the recipient and another part contains the data for the file.

The first of a multipart message is called part 0, and contains the main header block. This is what defines the headers that you are most familiar with, such as "From", "To" and "Subject". The body of this message part is typically a plain text message that indicates that this is a multipart message. This is done for the benefit of older mail clients that cannot parse MIME messages correctly. Next part, part 1, typically contains the actual body of the message that would be displayed by the mail client. Additional parts may contain file attachments and other information. In the case of a multipart message that contains both plain and styled HTML text versions of a message, part 1 is typically the plain text version of the message while part 2 contains the HTML version. The mail client can then make a decision based on its own configuration as to which version of the message it displays.

Part
This property returns the current message part index. All messages have at least one part, which consists of one or more header fields, followed by the body of the message. The default part, part 0, refers to the main message header and body. If the message contains multiple parts (as with a

message that contains one or more attached files), this property can be set to refer to that specific part of the message.

PartCount
This property returns the number of parts in the current message. All messages have at least one part, referenced as part zero. Multipart messages will consist of additional parts which may be accessed by setting the Part property.

CreatePart
Create a new, empty message part. If the message was not originally a multipart message, it will be restructured into one. Otherwise, the new part is simply added to the end of the message. This method will cause the current message part to change to the new part that was just created.

DeletePart
Delete the message part from the message. If the message part is in the middle of the message, it will cause the subsequent parts of the message to be reordered. You should not delete part zero to delete a message; use the **MimeDeleteMessage** method instead.

Importing Messages

The control can be used to import existing messages, either from memory or from a file. Once the message has been parsed, the application can examine or modify specific parts of the message. The following methods are provided to import the contents of a message:

ImportMessage
The simplest method of importing a message, this method reads the contents of the specified file and imports it into the current message. This method is typically called immediately after **MimeCreateMessage** to load a file into a new message context.

Exporting Messages

After a message has been created or modified, it can be exported to a file or to memory. Exporting the message to a memory buffer is particularly useful when using the control with another one of the SocketTools libraries. For example, the contents of a message can be exported to memory, and that memory address can be passed to the Simple Mail Transfer Protocol (SMTP) control for delivery to the recipient. The following methods are provided to export the contents of a message:

ExportMessage
This method exports the current message to a file. When using this method, only certain headers are exported and they may be reordered. To force all headers to be included in the message or to preserve the order of the headers, set the Options property.

File Attachments

In addition to simple text messages, one or more files can be attached to a message. The process of attaching a file involves creating a multipart message, encoding the contents of the file and then including that encoded data in the message. The following methods are provided to manage files attached to the message, as well as attach files to an existing message:

Attachment
A property which returns the name of a file attachment in the current message part. This property serves two purposes, to determine if the current message part contains a file attachment, and if so, what file name should be used when extracting that attachment.

AttachFile
This method attaches the contents of the file to the message. The file will be attached using the specified encoding algorithm and will become the current message part. If the message is not a multipart message, it will be converted to one; if it already is a multipart message, the attachment will

be added to the end of the message.

AttachData
This method works in similar fashion to **MimeAttachFile**, except that instead of the contents of a file, the data in a memory buffer will be attached to the message. If the message is not a multipart message, it will be converted to one; if it already is a multipart message, the attachment will be added to the end of the message.

AttachImage
This method attaches an inline image file to the message. It is similar to the **AttachFile** method, except that the image is designed to be referenced as an embedded graphic in an HTML message. This method will automatically set the correct header values for an inline image attachment, and enables the developer to specify a content ID which is used in the HTML message.

ExtractFile
Extract the file attachment in the current message part, storing the contents in a file. The attachment will automatically be decoded if necessary. This method also recognizes uuencoded attachments that are embedded directly in the body of the message, rather than using the standard MIME format.

Mail Addresses

The Mail Message control also has methods which are designed to make it easier to work with email addresses. Addresses are typically in the format of "user@domain.com" however additional information can be included with the address, such as the user's name or other comments that aren't part of the address itself. The control can parse these addresses for you, returning them in a format that is suitable for use with other protocols such as the SMTP control.

ParseAddress
Parse an email address that may include an address without a domain name or comments in the address, such as the user's name. For example, the From header field may return an address like "Joe Smith <joe@example.com>"; this method would parse the address and return "joe@example.com", the actual address for the user.

Recipient
It is common for certain headers to contain multiple addresses separated by a comma. These addresses may also include comments such as the user's name. This property array returns a list of valid addresses defined in the current message. For example, the To header field may contain "Tom Jones <tom@example.com>, Jerry Lewis <jerry@example.com>"; this property array would return "tom@example.com" and "jerry@example.com" as the two addresses listed. The total number of addresses that are available is returned by the Recipients property.

Message Storage

The control has a collection of methods which makes it simple for an application to store a group of messages together in a single file, search for and retrieve specific message. The collection of messages is referred to as a "message store" and messages may either be stored in a plaintext format or in a compressed binary format.

OpenStore
This method is used to open an existing message store or creates a new storage file. If a storage file has been opened previously, it will be closed and the new storage file will be opened. The storage files may either be plaintext, or stored in a compressed format. It also supports opening storage files in the UNIX ***mbox*** format.

StoreSize
This property returns the total number of messages that currently in the message store, including deleted messages. Each message is referred to by an integer which is its index into the storage file.

StoreIndex

This property specifies the current message index into the storage file. Messages are identified by an integer value that starts at one for the first message and increments for each additional message in the storage file. If no message store has been opened, this property will return a value of zero. Changing the value of this property changes the current message index for the message store.

FindMessage

An application can search the message store for messages that match any header value. Searches can be complete or partial, and may be case-sensitive or case-insensitive. For example, this method can be used to enumerate all of the messages in the storage file that were sent by a specific user or match a specific subject.

ReadStore

This method reads a message from the storage file and replaces the current message. If the application modifies the message, it can replace the message in the storage file or discard the changes.

WriteStore

This method writes the current message to the message store. Note that the message store must be opened for write access, and the message will always be appended to the storage file. The **StoreIndex** property is updated with the index value for the new message.

DeleteMessage

This method flags a message for deletion from the message store. Once a message has been flagged for deletion, it may no longer be accessed by the application. When the storage file is closed, the contents of the deleted message will be removed from the file.

ReplaceMessage

This method replaces an existing message in the storage file, overwriting it with the current message. Unlike many of the other methods which do not permit the application to reference a deleted message, this method can be used to replace a previously deleted message.

CloseStore

The message store must be closed when the application has finished accessing it. This method updates the storage file with any changes, purges all deleted messages and closes the storage file. If the storage file is locked for exclusive access, this method will release that lock, allowing another process to open the file.

Network News Transfer Protocol

The Network News Transfer Protocol (NNTP) control enables applications to access a news server, list the available newsgroups, retrieve articles and post new articles. It is common for this control to be used in conjunction with the Mail Message control to construct the articles, since a news article uses the same general format as an email message.

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
Establish a connection to the NNTP server. Once the connection has been established, the other methods in the control may be used to access the newsgroups and/or post new articles to the server.

Authenticate
Provide a user name and password to authenticate the client session. This should only be used if required by the server. Not all news servers require authentication, and some only require authentication when posting articles. If you attempt to perform a function that requires authentication, an error will be returned that indicates you should authenticate and then retry the operation.

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

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.

Newsgroups

News articles are posted in hierarchical groups, similar to how files are stored in folders. Each level in the newsgroup hierarchy is separated by a period, so newsgroup names look like microsoft.public.vc. This is Microsoft's newsgroup for articles about Visual C++ programming. Additional subgroups are used to further narrow the topic; for example, there's the microsoft.public.vc.3rdparty newsgroup for third party tools and components for Visual C++, and the microsoft.public.vc.atl newsgroup which discusses issues related to the Active Template Library. The NNTP control provides the following methods for accessing newsgroups on the server:

ListGroups
This method requests that the server return a list of all of the newsgroups that are available. The application can also request that only groups which were created since a specific date should be returned. This allows the application to maintain a list of newsgroups on the local system, and then use this method to periodically update that list based on the date it was last modified. If the method is successful, the application should call the **GetFirstGroup** method to begin processing the group list.

OnNewsGroup
This event is generated for each newsgroup returned by the **ListGroups** method. Information about the group is passed to the event handler as arguments to the event, including the name of the group, the first available article number and the last available article in the group.

SelectGroup
This method is used to select a newsgroup as the current group. Once selected, the application has access to the articles in that newsgroup.

News Articles

News articles are the messages posted to one or more newsgroups. Articles are referenced by their article number, which is a value assigned by the news server. These articles have a structure that is the same as an email message, with some slightly different headers. Because of this, you can use the Mail Message interface to parse articles that you retrieve, as well as create new articles to post to the server. The following methods are used to access and create news articles:

ListArticles
This method requests that the server return a list of articles that are available in the current newsgroup. The application can request that all articles be returned, or only those articles which fall into a certain range of article numbers.

OnNewsArticle
This event is generated for each article returned by the **ListArticles** method. Information about the news article is passed to the event handler as arguments to the event, including the article ID, size, subject, author and date that the article was posted.

GetArticle
Retrieve an article from the server, storing the contents in a string buffer or byte array. This can be used to process the contents of an article without the overhead of storing it in a file on the local system.

StoreArticle
Retrieve an article from the server and store it in a file on the local system.

PostArticle
This method posts an article to one or more newsgroups on the server. A newsgroup article is similar to an email message, and the MIME interface may be used to create the article headers and body. One important difference is that the message must contain a header named "Newsgroups" with the value set to the newsgroup or newsgroups that the article should be posted to; multiple newsgroups should be separated by commas. If this header is not defined, the posting will be rejected by the server and the method will return an error. You should also be aware that some servers limit the number of newsgroups that a message can be posted to. When an article is posted to more than one newsgroup at a time, this is called cross-posting. Current convention says that an article should not be cross-posted to more than five newsgroups at a time. Also keep in mind that multi-posting (posting the same article to different newsgroups separately) is generally discouraged and should never be done on USENET.

Attaching Files

It is possible to attach files to newsgroup articles; however it should only be done if it is considered appropriate for the group. Many newsgroups have their own acceptable use policies which determine whether or not file attachments, particularly large binary files, are acceptable. If the newsgroup accepts attachments, you can use one of several methods for posting files. It is recommended that you use the File Encoding control to handle the actual encoding of the data.

Uuencode
A uuencoded file attachment is included directly in the body of the message. Because the MIME interface creates a multipart message even when uuencoding is specified, the File Encoding control should be used to encode the data and then it should be included in the main body of the message.

Base64

A Base64 file attachment has the same structure as what is used by email messages. This requires that a multipart message be created, with the encoded data attached as a part of the message. You can use the MIME control to create this kind of message. Note that not all third-party newsreaders correctly handle multipart messages.

yEnc

An encoding method commonly used on USENET, it's similar to uuencoded attachments where the file data is part of the body of the message. The File Encoding control should be used to encode the data and then it should be included in the main body of the message. More information about yEnc encoding can be found at www.yenc.org

News Feed Control

Really Simple Syndication (RSS) is a collection of standardized formats that are used to publish information about content that is frequently changed. A news feed is published in XML format, which contains one or more items that includes summary text, hyperlinks to source content and additional metadata that is used to describe the item. News feeds can be used for a variety of purposes, including providing updates for weblogs, news headlines, video and audio content. RSS can also be used for other purposes, such as a software updates, where new updates are listed as items in the feed.

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

The first step your application must take is to initialize the library, which will load the required system libraries and initialize the internal data structures that are used. If the control is placed on a form, then the container automatically handles the initialization of control. If the control is created dynamically, then your application is responsible for initialization.

Initialize
Initialize the control and load the Windows Sockets control for the current process. If the control is created dynamically using a method 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.

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.

News Channels

A news feed consists of a channel that contains each of the news feed items. The following properties and methods are used to access and manage the news feed channel:

Description, Title
These properties return strings that provide an overview of the news feed and a title that should be displayed for the feed. The text returned by the **Title** property is typically brief, while the text returned by the **Description** property can be more verbose, such as a paragraph that summarizes the content of the feed.

LastBuild, Published
These properties return date and time values that identify when the feed was last modified, and when it was published. While these values are often the same, they can be different and some feeds do not specify them at all. The **Published** property is used to specify the date when the feed was first published with the current collection of news items. If the content of a news item subsequently changes, but no new items have been added, then the publish date may remain the same but the **LastBuild** property would be changed to reflect when the content was modified.

Open
Open the channel by specifying a URL to the resource that contains the news. The URL can identify a remote feed that is downloaded using the HTTP or HTTPS protocols, or it can be a file on the local

system or network.

Parse

Parse a string buffer that contains a news feed. This function is similar to the **Open** method, however it used to parse a string that contains the news feed. This method would typically be used when the feed content is obtained from a different source, such as a database or by using a different protocol. For example, the news feed could be downloaded using the **FtpClient** control and then passed to this function.

Close

This method closes the feed that was opened by a previous call to the **Open** or **Parse** method. When the information in a news feed is no longer needed, this method will release the resources allocated to process the feed. The current news feed will automatically be closed when an instance of the control is destroyed.

Store

This method is used to store a news feed as a file on the local system. This is typically used to cache the contents of a news feed or to track the changes made to the feed over time. It is recommended that the application periodically check the publication date of the feed to ensure that they have current version.

News Items

News feed items are identified by a numeric value called the item ID. This is used with other functions to return information about a specific news item. The first item in a news feed has an ID of one and it increments for each additional item in the feed.

ItemCount

This property will return the number of news items in the feed. The first item in the feed has a value of one, and it increments for each additional news item. This property can be used in conjunction with the **GetItem** method to enumerate all of the news items in the feed.

ItemTitle

This property will return a string which specifies a title for the news item. If no title has been specified, this property will return an empty string. Although it is not required for a news item to have a title, a feed that conforms to the standard must have either a title or a description of the item, which is returned by the **ItemText** property

ItemText

This property will return a string that contains a summary of the current news item. This may property may return either plain text or HTML formatted text. If no text has been specified for the current item, this property will return an empty string. Although it is not required for a news item to have a description, a feed that conforms to the standard must have either a description of the item or a title, which is returned by the **ItemTitle** property.

ItemLink

This property will return a string which specifies a URL that provides additional information about a news item. If the news item summarizes the contents of an article, this property typically provides a link to the complete article. If a link is not specified for the news item, this property will return an empty string.

ItemGuid

This property will return a string which uniquely identifies the current news item. If this property is defined, it is guaranteed to be a unique, persistent value. It is important to note that this string does not have to be a standard GUID reference number, it can be any unique string. If there is no unique identifier associated with the current item, this property will return an empty string.

GetItem

This method is used to return information about a specific news item based on the item ID. This becomes the current news item, and it updates the item-related properties such as **ItemTitle** and **ItemText**.

FindItem

This method is used to search the feed channel for a specific item, based either on its GUID, title, link or publication date. When searching for a specific item, only searches by GUID are guaranteed to return a unique news item. However, since not all news feeds may provide GUIDs for their news items, additional search criteria can be used when necessary. If this method succeeds, the item that matched the search criteria becomes the current news item.

Post Office Protocol

The Post Office Protocol (POP3) control enables an application to retrieve a user's mail messages and store them on the local system. The control provides support for all of the standard functionality such as listing and downloading messages, as well as extended features such as the ability to retrieve only the headers for a message or just specific header values. The control also has methods for changing the user's password and sending messages if they are supported by the server.

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 method 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
Establish a connection to the POP3 server. Once the connection has been established, the other methods in the control may be used to access the resources on the server.

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

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.

Managing Messages

There are methods in the POP3 control for managing messages which enables the application to list, delete and retrieve messages stored on the server. Messages are identified by a number, starting with one for the first message in the mailbox. The most typical operation for a POP3 client is to retrieve each message, store it on the local system and then delete the message from the server. Any processing that is done on the message would then be done on the local copy.

Message
This property sets or returns the message number for the currently selected mailbox. Message numbers range from 1 through the number of messages available on the server, as returned by the **MessageCount** property.

MessageCount, LastMessage
A property which returns the number of messages available for retrieval. There are two values the application should use. One is the number of currently available messages and the other is the last valid message number. As messages are deleted from the server, the total number of available messages will decrease; however, the last available message number will remain constant.

MessageSize
This property returns the size of the message in bytes. One thing to be aware of when using this method is that some servers will only return approximate message sizes. In addition, because of the difference between the end-of-line characters on UNIX and Windows systems, the size reported by the server may not be the actual size of the message when stored on the local system. Therefore, the application should not depend on this value as an absolute. For example, it should not use this value to determine the maximum number of bytes to read from the server; instead, it should read until the

server indicates that the end of the message has been reached.

GetMessage
This method is used to retrieve a message from the server and copy it into a local string or byte array buffer. This method will cause the current thread to block until the message transfer completes, a timeout occurs or the transfer is canceled.

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

DeleteMessage
Mark the message for deletion. When the connection with the server is closed, the message will be removed from the user's inbox. An important difference between the POP3 and IMAP protocols is that when a message is marked as deleted on a POP3 server, that message can no longer be accessed. An attempt to retrieve a message after it has been marked for deletion will result in an error. The only way to undelete a message once it has been deleted is to terminate the connection with the server by calling the Reset method instead of calling the Disconnect method.

Message Headers

The POP3 control also includes methods which enable the application to access just the headers for a message. This can be useful if the program doesn't want to incur the overhead of downloading the entire message contents. The following methods can be used to examine the headers in a message:

GetHeaders
This method returns the complete set of headers for the specified message. If your program has to process multiple header fields, this is the most efficient method to use. It is possible to retrieve specific header values, however not all servers support that option and it is somewhat slower because it involves sending individual commands to request each value.

GetHeader
This method returns the value for a specific header field in a message. This method does not require that you parse the message headers; however it does incur additional overhead. It is also important to note that not all servers support the command that is used to request the header value. If this method fails with the error that the feature is not supported, you should use the **GetHeaders** method instead.

MessageUID
This property returns the unique ID (UID) that the server has associated with the message. The UID can be used by an application to track whether or not it has previously viewed the message. Unlike the message number, which can change between client sessions, the message UID is guaranteed to be the same value across sessions until the message is deleted.

Remote Command Protocol

The Remote Command protocol enables an application to execute commands on a server, with the output of the command returned to the client. The SocketTools control actually implements three related protocols: rexec, rshell and rlogin. The choice of protocols is determined by the port that is selected when a connection is established.

Rexec
The rexec protocol enables a client application to execute a command on a server. Output from the command is returned to the client and the connection is closed when the command terminates. The client connects on port 512 and must provide a user name and password to authenticate the session.

Rshell
The rshell protocol is similar to rexec in that it enables a client to execute a command on a server. Output from the command is returned to the client and the connection is closed when the command terminates. The client connects on port 514 and must provide a user name. The primary difference between the rexec and rshell protocols is that rshell does not require a password. Instead, it uses what is called "host equivalence" to determine if the client is permitted to execute commands as that user. On a UNIX based operating system, host equivalence is controlled by the /etc/hosts.equiv and the .rhosts file in the user's home directory. These files list the host names and user names which are permitted to execute commands using the rshell protocol. Consult your operating system manual pages for more information about how to configure host equivalence.

Rlogin
The rlogin protocol is similar to Telnet in that it provides an interactive terminal session. The connection is closed when the user logs out or the shell process on the server is terminated. The client connects on port 513 and must provide a user name and terminal type. If there is an entry in the host equivalence tables for the user and local host, then the client will be automatically logged in and provided with a shell prompt. If there is no host equivalence, the client will be prompted for a password. The terminal emulation control can be used to provide ANSI or DEC VT-220 emulation services if needed.

An important consideration when deciding whether to use rexec, rshell or rlogin is how the server is configured and the type of command being executed. If there is no entry for the local host in the server's host equivalence tables, then the rexec command should be used instead of rshell.

When using rexec or rshell, it is important to keep in mind that although the command is executed with the privileges of the specified user, that user is not actually logged in. The user's login script is not executed and the program will not inherit the user's normal environment as it would during an interactive session. If you are connecting to a UNIX system, you should not attempt to execute programs which try to put standard input into raw mode; an example of this would be the vi editor. If you are connecting to a Windows system, you should not execute a program which uses a graphical interface. Only programs which read standard input and write to standard output are suitable for use with rexec or rshell.

The following methods 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 method 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.

Execute
Execute the specified command on the server. The rshell or rexec protocol is selected based on the

port number that is specified. Output from the command will be returned to the client to be read. When the command terminates, the connection to the server will be closed.

Login

Establish an interactive login session which is similar to how the Telnet protocol works. If there is no host equivalence with the local host, you will be prompted for a password. Output from the session will be returned to the client, and when the client logs out the connection will be closed.

Read

Read the output generated by the command. Your application would typically call this method in a loop until all of the data has been read or an error occurs.

Search

Search for a specific sequence of characters in the output returned by the server. The method returns when the sequence is encountered or when a timeout occurs. The data captured up to the point where the character sequence was matched is returned to the caller for processing.

Disconnect

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

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.

Secure Shell Protocol

The Secure Shell (SSH) protocol enables an application to establish a secure, interactive terminal session with a server, or execute commands remotely on the server, with the output of the command returned to the client. The SocketTools ActiveX control supports both version 1.0 and 2.0 of the protocol.

The following methods 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 method 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
Establish a connection to the server. Once the connection has been established, the other methods in the control may be used to interact with the server.

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

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

Once connected to the server, any output generated by the command shell or a program executed on the server will be sent as data for the client to read. Any input to the program is sent by the client and received and processed by the server. The following methods are used:

Read
Reads any output that has been generated by the program executing on the server. If the server 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 read.

ReadLine
The ReadLine method reads data from the server 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.

Peek
The Peek method can be used to examine the data that is available to be read from the internal receive buffer. If there is no data in the receive buffer at that time, a value of zero is returned. The Peek method will never cause the client to block, and so may be safely used with asynchronous connections.

Write
Send data to the server which will be received as input to the program. If the method succeeds, the return value is the number of bytes actually written. This method should always be used when sending binary data to the server.

WriteLine
The WriteLine method writes a line of text to the server and terminates the line with a carriage-return

and linefeed control character sequence. Unlike the Write method which sends arbitrary bytes of data to the server, this method is specifically designed to send a single line of text data from a string.

Command Processing

The SSH protocol can be used to connect to a server, log in and execute one or more commands, process the output from those commands and display it to an end-user using a graphical interface. The user never sees or interacts with the actual terminal session. The control interface provides methods which can simplify this kind of application, reducing the amount of code needed to process the data stream returned by the server.

Execute

This method executes a command on a server and copies the output to a user-specified buffer, with the exit code for the remote program as the method's return value. This is a convenience method that enables you to execute a remote command in a single call, without having to write the code to establish the connection and read the output.

Search

This method is used to search for a specific character or sequence of characters in the data stream returned by the server. The control will accumulate all of the data received up to the point where the character sequence is encountered. This can be used to capture all of the output from a command, or search for specific results returned by the command as it executes on the server.

Simple Mail Transfer Protocol

The Simple Mail Transfer Protocol (SMTP) enables applications to deliver email messages to one or more recipients. The control provides an interface for addressing and delivering messages, and extended features such as user authentication and delivery status notification. This control is typically used in conjunction with the Mail Message control to create the messages, and the Domain Name Service control to determine what servers are responsible for accepting mail for a specific user.

Mail Exchanges

When a message is delivered to a user, the application must determine what mail server is responsible for accepting messages for that user. This can be accomplished using the Domain Name Services (DNS) protocol, a protocol that is most commonly used to resolve host names such as www.microsoft.com into Internet addresses. This is typically accomplished by sending a request to a nameserver, a computer system that provides domain name services. In addition to resolving host names, nameservers can also provide information about those servers which are responsible for accepting mail for a given domain. There can be multiple servers which process mail for a domain with each server assigned a priority as part of their mail exchange (MX) record. If there is no mail exchange record for a domain, then the domain name itself is used.

To deliver a message directly to the recipient, you must examine the recipient address and request the list of mail exchanges for that user's domain. Using the DNS control, this is done by reading the **MailExchange** property array. If the recipient address is joe@example.com, you would want to enumerate the mail exchanges for the example.com domain. This will give you the name of the servers that will accept mail for users in that domain. For example, the property may return the host name mail.example.com as the name of the server which will accept mail for users in the example.com domain. Note that it is possible that one or more of the mail exchanges for a domain may not be in the recipient domain itself. In other words, it is possible that smtp.othercorp.net could be returned as a mail exchange for example.com. This is frequently the case when another organization is forwarding mail for that domain.

Therefore, there are three general steps that you must take when delivering mail directly to the recipient:

1. Parse the address of each recipient in the message. If you are using the MIME control, the Recipient and Recipients properties can be helpful in extracting all of the recipient addresses. Everything after the atsign (@) in the address is the domain portion of that address.

2. Perform an MX record lookup using the DNS control by setting the **HostName** property to the domain name and reading the values returned in the **MailExchange** property array. This property will return the name of the servers responsible for accepting mail for that user. If there are more than one server, they will be returned in order of their relative priority, with the highest priority server having a lower index value. This means that you should attempt to connect to those servers in the order that they are returned by the property, starting with an index value of zero.

3. Attempt to connect to the first server returned by the **MailExchange** property array. The connection should be on the default port, and you should not attempt to use any authentication. If the server accepts the connection, then use the **SendMessage** method to deliver the message. If the connection is rejected or the message is not accepted, attempt to connect to the next mail exchange server until all servers have been tried.

4. If no mail exchange servers were returned by the MIME control's **MailExchange** property, or you could not connect to any of them, attempt to connect to the domain specified in the address using the default port. If the connection succeeds, then deliver the message. If you cannot connect or the message is not accepted, then report to the user that the message could not be delivered.

One last important consideration is that many Internet Service Providers now block outbound connections on

port 25 to any mail servers other than their own. If you are unable to establish any connections, either with the error that the connection was refused or it consistently times out, contact your ISP to determine if port 25 is being blocked as an anti-spam measure. If this is the case, it will be required that you relay all messages through their mail servers.

Relay Servers

In some situations it may not be possible to send mail directly to the server that accepts mail for a given domain. The two most common situations are corporate networks which have centralized servers that are responsible for delivering and forwarding messages, or an Internet Service Provider (ISP) which specifically blocks access to all mail servers other than their own. This is usually done as either a security measure or as a means to inhibit users from sending unsolicited commercial email messages. If the standard SMTP port is being blocked, then any connection attempts will either fail immediately with an error that the server is unreachable, or the connections will simply time-out. In either case, a relay server must be specified in order to send email messages.

A relay server is a system which will accept messages addressed to users who may be in a different domain, and will relay those messages to the appropriate server that does accept mail for the domain. Using a relay server is generally easier than sending messages directly to the recipient. In order to send a message through a relay, you need to perform the following steps:

1. Connect to the relay server as you would normally.
2. Authenticate the client to the server. This may or may not be required, depending on how the server is configured. Some servers may be configured to only require authentication if you are connecting from an IP address that is not recognized as part of that system's network, for example, if you are connecting using a different Internet Service Provider. Others may always require authentication. Check with the server administrator if necessary to determine if and when authentication is required.
3. Use the **SendMessage** method to deliver the message to the recipients through the relay server. If there are multiple recipients, you can use the MIME control to enumerate the recipient addresses and then pass them to the **SendMessage** method.

It is important to note that using a mail server as a relay without the permission of the organization or individual who owns that server may violate Acceptable Use Policies and/or Terms of Service agreements with your service provider. Systems which relay messages from anyone, regardless of whether the message is coming from a recognized domain, are called open relays. Because open relays are often used to send unsolicited email, many administrators block mail that comes from one. It is recommended that users check with their network administrators or Internet service providers to determine if access to external mail servers is restricted and what is the acceptable use policy for relaying messages through their mail servers.

The following methods are available for use by your application:

Initialize

Initialize the control and load the Windows Sockets control 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 method 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

Establish a connection to the SMTP server. Once the connection has been established, the other methods in the control may be used to deliver messages to the server.

Authenticate

Authenticate yourself to the server using a username and password. This method should be called

immediately after the connection has been established to the server. This is typically required if you are attempting to use the mail server as a relay, asking it to forward the message on to the server that actually accepts email for the recipient. Many Internet Service Providers (ISPs) require that users authenticate prior to sending mail through their servers. You may need to contact the server administrator to determine if authentication is required.

Disconnect

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

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.

Message Delivery

There are two general methods that can be used to deliver messages through the mail server. In most cases, it can be done with a single method call. However, there are some circumstances where it would be more appropriate to perform the transaction in stages. The SMTP control supports both methods.

SendMessage

This is the simplest method for sending an email message through the server. You provide the sender and recipient addresses, along with the message contents and the method will submit the message to the server for delivery.

CreateMessage

This method begins a transaction in which a message is dynamically composed, addressed and delivered in stages. You provide the sender address and message size to this method, and after it returns you begin the next stage, which is addressing the message.

AddRecipient

This method adds a recipient address to the recipient list for the message. This should be called once for each recipient, as well as for any recipients who are to receive "blind copies" of the message. A blind copy is when the message is sent to a recipient, but that recipient's address is not listed in any of the headers of the message; the other recipients will be unaware that the message was delivered to him. Most servers have a limit of approximately 100 recipients per message. It is possible that this method will return an error for a specific recipient address; the address may be malformed or it may not be acceptable for some other reason. This does not mean that the message will be rejected in its entirety, only that the specified recipient is not acceptable.

AppendMessage

This method should be called after all of the recipients have been added. It is used to send the contents of the message to the server. It is also possible to use the lower level Write method to send data directly to the server, however **AppendMessage** is generally easier to use and can write data from memory, the system clipboard or from a file on disk.

CloseMessage

This method is called after the entire message has been sent to the server. This terminates the transaction and the message is submitted for delivery. Note that it is possible for the server to accept the message up to this point and then reject it at this final step due to some restriction, such as the message being too large.

SocketWrench

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. 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 server, 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 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 server 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 read. This method should always be used when reading binary data from the server into a byte array.

ReadLine

Read a line of text from the socket, up to an end-of-line character sequence or when the server closes the connection. This method is useful when the client and server are exchanging textual data, as is common with most command/response application protocols.

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. This method should always be used when sending binary data to the server.

WriteLine
Write a line of text to the socket, terminating it with an end-of-line character sequence. This method is useful when the client and server are exchanging textual data, as is common with most command/response application protocols.

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. If the property returns a value of True, the **Read** method will return without causing the application to block. If the property returns False, there is no data available to 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 server 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 server 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 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.

Telnet Protocol

The Telnet Protocol control enables an application to connect to a Telnet server, which provides an interactive terminal session similar to how character based consoles and terminals work. The user can login, enter commands and interact with applications programmatically or in conjunction with the terminal emulation control.

The following methods 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 method 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
Establish a connection to the server. Once the connection has been established, the other methods in the control may be used to interact with the server.

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

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

Once connected to the Telnet server, any output generated by a program on the server will be sent as data for the client to read. Any input to the program is sent by the client and received and processed by the server. The following methods are used:

Read
Reads any output that has been generated by the program executing on the server. When the client first connects, the server typically executes a login program that requests the users authenticate themselves by entering a user name and password. Once the user has logged in, they are usually given a command line prompt where they can enter commands to be executed on the server. If the server closes the connection, the Read method will indicate that with an error result and the client can disconnect from the server at that point.

Write
Send data to the Telnet server which will be received as input to the program. If the local echo option is enabled, then the client is also responsible for writing the input data to the display device, if there is one. If local echo is not enabled, the server will automatically echo back any characters written as data to be read by the client.

Telnet Modes

Telnet supports several modes of operation and the option negotiation phase, which occurs when a connection is established, is handled automatically by the control. There are two key modes which affect how the client session works:

Binary
If this property is set to True, the data between the client and server is not buffered and the high bit is

not removed from any characters. If the application is executing a program which uses text mode windowing features (i.e.: it draws boxes on the display) then this mode must be enabled to ensure that the client processes the data correctly and it isn't buffered a line at a time. If this mode is disabled, then the data exchanged between the client and server will be buffered a line at a time and any 8bit characters will be stripped. This mode is enabled by default.

LocalEcho

If this property is set to True, it is the responsibility of the client to echo any data that it is sending to the server. For example, if the character "A" is sent to the server, the application must also send the character "A" to whatever interface the user is interacting with, such as a terminal emulation window. The default mode is for this option to be disabled, which means that the server will echo back any data that is sent to it.

Command Processing

The Telnet protocol can be used to connect to a server, log in and execute one or more commands, process the output from those commands and display it to an end-user using a graphical interface. The user never sees or interacts with the actual terminal session. The Telnet interface provides methods which can simplify this kind of application, reducing the amount of code needed to process the data stream returned by the server.

Login

This method is used to automatically log a user in, using the specific user name and password. This method is specifically designed for UNIX based servers or Windows servers which emulate the same basic login sequence.

Search

This method is used to search for a specific character or sequence of characters in the data stream returned by the server. The control will accumulate all of the data received up to the point where the character sequence is encountered. This can be used to capture all of the output from a command, or search for specific results returned by the command as it executes on the server.

Terminal Emulation

The Terminal Emulation control provides a virtual terminal interface for emulating an ANSI or DEC VT-220 compatible character-based terminal. It can be used in conjunction with the Telnet interface or the Remote Command interface to display the output of commands executed on a server. It can also be used independently of any other networking control, such as providing emulation services for a serial connection.

Display Management

The control provides a number of properties and methods to manage and update the virtual display. The most commonly used methods are:

BackColor

This property can be used to change the background color displayed by the virtual terminal.

ColorMap

This property array can be used to change the default colors which are used when escape sequences are used to change the foreground or background color of a character cell. In most cases the default color map will be appropriate, but applications can change the RGB values associated with an entry in the color map if needed. For example, the default value for the color gray is at position 8 in the color map index with an RGB value of 192,192,192. If you wanted to use a darker color, you could change the RGB value to 128,128,128

Emulation

This property specifies the type of emulation that will be performed by the control. The control is capable of emulating an ANSI console, a DEC VT-100 and DEC VT-220/320 terminal.

FontName

This property sets the name of the font which is used by the control to draw text on the display window. Note that there is also a Font property which returns a Font object for more control over the font used by the emulator. It is recommended that you only used fixed-width fonts such as Terminal or Courier New.

FontSize

This property sets the size of the font which is used by the control to draw text on the display window. Note that there is also a Font property which returns a Font object for more control over the font used by the emulator.

ForeColor

This property can be used to change the foreground color displayed by the virtual terminal.

Write

This is the most commonly used method of writing to the display. This method will automatically parse the data being written for escape sequences and update the display appropriately.

Refresh

Refresh the virtual display, updating the current cursor position and caret. The control will periodically refresh the display automatically based on its own internal state, but the application can call this if it wishes to force the display to refresh at that time.

Reset

This method can be used to reset the display window, the font being used and the size of the display. Note that resetting the display causes the contents of the display to be cleared.

Cursor Control

There are a number of properties and methods which enable an application to have direct control over cursor positioning, clearing the display and so on. In most cases these methods are called automatically by the control

as the result of processing the escape sequences found in the data being written to the display. However, an application can choose to manage the display itself. One important thing to keep in mind is that the X,Y positions used by these properties and methods refer to the cursor position in the virtual display and correspond to columns and rows, not pixels.

There is also a slight difference in terminology that you should be aware of when reading the technical reference documentation. In Windows, the term "cursor" is typically used to refer to the mouse pointer, while "caret" is used to refer to the blinking marker that is displayed at the current position in the display. In the documentation for the emulator, the term "cursor" is used in the same way that it is used for character based terminals, as the marker for the current position in the display. Therefore, in terms of the control, you can think of the cursor and the caret as being synonymous.

CursorX
This property returns the current position of the cursor in the display, or can be used to change the current position. The current position is given in columns and indicates where the next text character will be displayed.

CursorY
This property returns the current position of the cursor in the display, or can be used to change the current position. The current position is given in rows and indicates where the next text character will be displayed.

Clear
This method clears the contents of the display. You can clear from the start of the display to the current cursor position, from the current position to the end of the display or the entire display.

DelLine
This method deletes the line at the current cursor position, shifting the remaining lines in the display up.

InsLine
This method inserts a blank line at the current cursor position, shifting the following lines down.

ScrollDown
This method scrolls the display down by one line.

ScrollUp
This method scrolls the display up by one line.

Function Key Mapping

Another aspect of terminal emulation is how function keys and other special keys are handled by the application. The emulation control can be used to convert Windows virtual key codes into the escape sequences that are generated by character based terminals.

KeyMap
This property array allows the application to define character sequences that should be mapped to special keys. When a special key is pressed in the emulation window and there is an entry for it in the key map, the **KeyMapped** event is fired. For example, if the user presses the F1 key on the keyboard, the control will translate that key code into the three characters escape sequence ESC O P (the ASCII codes 27, 79, 80). That sequence of characters should be sent to the server, which will recognize it as the F1 function key being pressed. It is important to note that the different emulation types have different key mappings. Therefore, the server must be set to recognize the same type of terminal that you are emulating. If you have the emulation set as VT-220 but the server thinks that you are emulating a VT-100, it will not recognize some of the escape sequences correctly.

KeyMapped

This event is generated when the user presses a special key while the emulation window has focus, and that key is mapped to a string using the **KeyMap** property array. Typically an application will use this event to send the mapped key escape sequence to a server, such as a Telnet server.

Text Message Control

Short Message Service (SMS) is a text messaging service used by mobile communication devices to exchange brief text messages. Most service providers also provide gateway servers that can be used to send messages to a wireless device on their network using standard email protocols. The Text Message control provides methods that can be used to determine the provider associated with a specific telephone number and send a text message to the device using the provider's mail gateway.

The first step your application must take is to initialize the library, which will load the required system libraries and initialize the internal data structures that are used. If the control is placed on a form, then the container automatically handles the initialization of control. If the control is created dynamically, then your application is responsible for initialization.

Initialize

Initialize the control and load the Windows Sockets control for the current process. If the control is created dynamically using a method 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.

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.

Text Messages

Sending a text message is done with a single method call with several optional parameters. By default, messages are sent via an SMTP gateway, however the control was designed to be extensible so that additional methods could be integrated into future versions of the controls. For example, a third-party company may offer a service that allows messages to be sent using HTTP and that can be added as an additional service type.

Message

This property is used to specify the current message text. In most cases, a message should not exceed 160 characters in length, although some service providers may accept longer messages. If a message exceeds the maximum number of characters accepted by a service provider, the message may be ignored or it may be split into multiple messages.

PhoneNumber

This property is used to specify the current phone number for the device you want to send a message to. This can be a standard E.164 formatted phone number or an unformatted number. Any extraneous whitespace, punctuation or other non-numeric characters in the string will be ignored.

Sender

This property is used to specify the email address of the sender when the SMTP service is used to send the message. For other service types, this property typically specifies the phone number or shortcode associated with the sender.

Urgent

This property specifies whether a message will be flagged as urgent or not. If this property is set to True, the message will sent with a high priority. Note that this does not guarantee the message will be received any differently than a standard text message. Each wireless service provider may handle urgent messages differently, and some providers may simply ignore the message priority.

SendMessage

This method is used to send a text message. It accepts one or more optional parameters that can specify the sender, recipient and content of the message, or if no parameters are specified, will use the

PhoneNumber, **Sender** and **Message** property values.

Relaying Messages

When a text message is sent using the SMTP service, the default action is to attempt to connect directly to the wireless service provider's gateway server. However, many residential Internet service providers (ISPs) do not permit their customers to connect to third-party mail servers and will block the outbound connection. Some wireless service providers may also reject messages that originate from residential IP addresses.

To resolve this issue, the developer should allow the user to specify an alternate mail server that will relay the message to the wireless service provider. For residential users, this will typically be the mail server provided by their ISP. For business users, this will usually be their corporate mail server. The **ServerName** and **ServerPort** properties are used to identify the relay server, and the **UserName** and **Password** properties provide the credentials to authenticate the client session.

Relay

This property is used to determine if the control will send the message directly to the wireless service provider's gateway server, or if the message will be relayed through another mail server.

ServerName

This property is used to specify the host name or IP address of the server that will relay the message.

ServerPort

This property is used to specify the port number that will be used to establish the connection with the relay server. For SMTP servers, this would typically either be port 25 or port 587.

UserName

This property is used in conjunction with the **Password** property to authenticate the client session. If the message is being sent using SMTP, this would typically be the user name or email address of the person sending the message.

Password

This property is used in conjunction with the **UserName** property to authenticate the client session. If the message is being sent using SMTP, this would be the password associated with the user name.

Service Providers

When a service provider is mentioned in the documentation, typically it is referring to the wireless service provider (also commonly called a "carrier") that is responsible for providing network access for the mobile device. These are identified by name, such as "Verizon Wireless" and "AT&T Mobility". The control has a built-in table of known providers in North America, and can return this information to your application. Note that in some cases, a service provider may also refer to a specific service used to send a text message.

Provider

This property the preferred wireless service provider associated with the current phone number. Changing the value of this property will change the preferred wireless service provider. If this property is an empty string, the default provider assigned to the recipient's phone number will be used.

ProviderCount

This property returns the number of wireless service providers supported by the control. This property is used in conjunction with the **ProviderName** property to enumerate all of the supported service providers.

ProviderName

This property array returns the name of supported wireless service provider and is used in conjunction with the **ProviderCount** property to enumerate all of the supported service providers. Typically this done to populate a user-interface control that enables the user to select a preferred service provider.

GetAddress

This method can be used obtain the email address for the wireless service provider's gateway that is associated with a phone number. This is done by sending an query to a server that will check the phone number against a database of known providers and the phone numbers that have been allocated for wireless devices.

GetProvider

This method can be used to obtain the name of the service provider associated with a phone number. This is done by sending an query to a server that will check the phone number against a database of known providers and the phone numbers that have been allocated for wireless devices.

Time Protocol

The Time protocol control enables an application to retrieve the current time from a server, and optionally synchronize the local system time using that value. The first step that your application must take is to initialize the control. After the control has been initialized, the application can request the current time from a system and update the local system clock if necessary.

Initialize

Initialize the control and load the Windows Sockets control 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 method 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.

GetTime

Return the current time from a server. The time and date retrieved from the server will be returned as a string formatted according to the user's current locale. If the date could not be retrieved, an empty string will be returned.

SetTime

Update the local system time with the value returned by **GetTime**. On Windows NT and later versions of the operating system, this method requires that the current user have the appropriate permissions to modify the system time or the method will fail.

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.

Time Conversion

The control also provides several properties which can be used to convert between the local date and time and UTC date and time for the value returned by the server. These properties are:

LocalDate

This property returns the network date and adjusts the value for the local timezone. The date is returned as a string formatted using the Short Date format for the current locale.

LocalTime

This property returns the network time and adjusts the value for the local timezone. The time is returned as a string formatted using the standard format for the current locale.

SystemDate

This property returns the network date in Coordinated Universal Time (UTC). The date is returned as a string formatted using the Short Date format for the current locale.

SystemTime

This property returns the network time in Coordinated Universal Time (UTC). The time is returned as a string formatted using the standard format for the current locale.

Web Location Control

The WebLocation control enables an application to obtain geographical information about the physical location of the computer system based on its external IP address. This can allow developers to know where their application is being used, and provide convenience functionality such as automatically completing a form based on the location of the user.

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

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

Methods

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

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.

Update

This method causes the control to update its various properties with information about the current location. The location service is queried to obtain current information about the physical location of the computer system based on its external IP address. The location data is cached and additional queries are only performed if it detects the external IP address for the local system has changed.

Uninitialize

Release all resources which 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.

Properties

The following properties provide information about the current location of the local computer:

Property	Description
ASNumber	An integer which is used to uniquely identify a global network (autonomous system) which is connected to the Internet. This value can be used to determine the ownership of a particular network.
CityName	A string which identifies the city at this location. These names will always be in English, regardless of the current system locale. If the city name cannot be determined, this member may contain an empty string.
Coordinates	A string which specifies the location expressed using the Universal Transverse Mercator

	(UTM) coordinate system with the WGS-84 ellipsoid. These coordinates are commonly used with the Global Positioning System (GPS).
CountryAlpha	A string which contains the ISO 3166-1 alpha-2 code assigned to the country. For example, the alpha-2 code for the United States is "US".
CountryCode	An integer value which identifies the country using the standard UN country code. For example, the numeric country code for the United States is 840.
CountryName	A string which contains the full name of the country in which the external IP address is located, such as "United States". These names will always be in English, regardless of the current system locale.
IPAddress	A string which contains the external IP address for the local system. If the system has been assigned multiple IP addresses, it reflects the address of the interface used to establish the connection with the location server.
Latitude	A real number which specifies the latitude of the location in decimal format. A positive value indicates a location which is north of the equator, while a negative value is a location which is south of the equator.
LocalTime	The current date and time at the location, adjusted for its time zone and whether or not it's in daylight savings time.
LocationId	A string which contains contains a string of hexadecimal characters which uniquely identifies the location for this computer system. This value is used internally by the location service, and may also be used by the application for its own purposes.
Longitude	A real number which specifies the longitude of the location in decimal format. A positive value indicates a location which is east of the prime meridian, while a negative value is a location which is west of the prime meridian.
Organization	A string which identifies the organization associated with the local system's external IP address. For residential end-users this is typically the name of their Internet Service provider, however it may also identify a private company.
PostalCode	A string which contains the postal code associated with the location. In the United States, this is a 5-digit numeric code. Local delivery portions of a postal code (such as the ZIP+4 code in the United States) are not included.
RegionCode	An integer which identifies the geographical region. This value corresponds to standard UN M49 region codes.
RegionName	A string which identifies a broad geographical area, such as "North America" or "Southeast Asia".
Subdivision	A string which identifies a geopolitical subdivision within a country. In the United States, this will contain the full name of the state or commonwealth. In Canada, this will contain the name of the province or territory.
SubdivisionCode	A string which is either a two- or three-letter code which identifies a geopolitical subdivision within the country. These codes are defined by the ISO 3166-2 standard. For example, the code for the state of California in the United States is "CA".
Timezone	A string which specifies the full time zone name. These names are defined by the Internet Assigned Numbers Authority (IANA) and have values like "America/Los_Angeles" and "Europe/London".
TzOffset	A integer which specifies the number of seconds east or west of the prime meridian

	(UTC). A positive value indicates a time zone which is east of the prime meridian and a negative value indicates a time zone which is west of the prime meridian.
TzShortName	A string which specifies the abbreviated time zone code. If daylight savings time is used within the time zone, then this value can change based on whether or not daylight savings is in effect. If a short time zone code cannot be determined, a value such as "UTC+9" may be returned, indicating the number of hours ahead or behind UTC.

Web Storage Control

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

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

Terminology

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

Application Identifiers

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

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

- MyCompany.MyApplication
- MyCompany.MyApplication.1

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

The AppId must only consist of ASCII letters, numbers, the period and underscore character. Whitespace characters and non-ASCII Unicode characters are not permitted. The maximum length of the string is 63 characters. It is not required for your application to create a unique AppId. Each storage account has a default internal AppId named `SocketTools.Storage.Default`. This AppId is used if a NULL pointer or an empty string is specified.

Containers

Storage containers are somewhat analogous to directories or folders in a file system, however they are general purpose and designed to allow you to control how your application accesses the data that's been stored. There are four container types which are defined by the control, and you can think of them as types of boxes or file cabinets which you store your data in.

It is important to keep in mind these containers are available to all users of your application, your program controls who has access to any particular data file. Your users will not be able to "browse"

any of the containers unless you specifically provide that capability by implementing it in your own code. There is no public access to any of the data which you upload, and our service does not use an open API accessible by third parties.

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

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

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

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

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

Objects

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

Labels

Object labels are similar to file names, and are a way to identify a stored object instead of using its internal object ID. However, there are some important differences. The most significant difference being labels are case-sensitive, unlike Windows file names. An object with the label "AppConfig" is considered to be different than one with the label "appconfig". Labels can contain Unicode characters,

but they cannot contain control characters.

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

Media Type

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

Initialization

The first step your application must take is to initialize the control and then open a storage container. The following methods 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.

Open

Opens a storage container for your application. Subsequent operations, such as storing, retrieving and copying objects will be performed within this container.

Close

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

Uninitialize

Release all resources which 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.

Data Storage

The control provides methods to upload and download to the storage container. You can store the contents of local files, or you can create objects from memory using strings or byte arrays.

GetData

Download object data and store it in a string or byte array provided by the caller.

GetFile

Download object data and store it in a file on the local system.

PutData

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

PutFile

Upload the the contents of a local file and store it as an object in the current container.

Data Management

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

FindFirst
Enables your application to search for and enumerate objects in a container based on their label and/or their media type. This method is used in conjunction with the FindNext method to list all matching objects in a container.

CompareData
Compares the contents of a string or byte array with the data stored in an object. This method can be used to determine if the contents of the buffer have changed since the data was previously stored using the PutData method.

CompareFile
Compares the contents of a local file with the data in a stored object. This method can be used to determine if the contents of a file have changed since it was previously stored using the PutFile method.

Copy
Copies the contents of a stored object to a new container, or duplicating the object within the same container using a different label.

Move
Moves the contents of a stored object to a new container.

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

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

DeleteAll
Deletes all objects which are stored in the current open container. This method resets the container back to its initial state, deleting all object metadata from the database and removing all stored data. This operation is immediate and the objects stored in the container are permanently deleted. They cannot be recovered by your application.

Other Methods

Several additional methods are available, allowing your application register and de-register custom application identifiers and validate object labels.

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

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

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

ValidateLabel

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

Whois Protocol

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

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 server, using either a host name or IP address. Once the connection has been established, the other methods in the control may be used to retrieve information from the server.

Search

This method submits a search keyword to the server. The keyword may specify a domain name, a user handle or a user mailbox, depending on the search type. Note that not all WHOIS servers support all search types. For example, many servers no longer support searching for user information based on email addresses.

Read

Read the data returned by the server, storing it in a string variable or byte array that is specified by the caller. This will contain the information about the domain specified when the Search method was called. Note that the data returned will typically be text, however it may not follow the same end-of-line conventions as Windows. For example, if the server is a UNIX or Linux system, the end-of-line may be indicated by a single linefeed, rather than a carriage-return/linefeed pair. Your application will have to account for this if the data is being displayed as-is to a user.

Disconnect

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

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.

SocketTools 11 Quick Start Guide

Quick Start Guide Overview

The Quick Start Guide is designed to help you get started quickly and easily using the SocketTools ActiveX controls. It is important to note that the topics will be general in nature, and more specific information about a particular component is available in the Technical Reference section. This guide is not meant to be a replacement for the either the Developer's Guide or the Technical Reference, and we encourage developers to review those sections thoroughly. Although this guide will cover the most common uses of the components, it is meant to serve as an overview; not every property, method and event will be discussed.

The examples provided in the Quick Start Guide presume some familiarity with the Visual Basic programming language. However, the basic concepts are the same regardless of what language is used. For information on using the controls in other languages, such as Visual C++, check the Language Support section of the Developer's Guide. For complete information on all of the properties, methods, events and constants used by the control, refer to the Technical Reference. Before performing any of the steps in this guide, you should have installed SocketTools on your development system.

To include any of the SocketTools controls in your project in Visual Basic, simply select the **Project**|**Components...**|**Controls** menu option and select the control that you wish to use. In other languages, follow the normal steps that are taken to include an ActiveX control in your development project.

Quick Start Concepts

Before getting started with using the controls, there are some general concepts used throughout the documentation which the developer should understand. If you are new to network programming, you are also encouraged to review the General Concepts section of 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. In SocketTools, 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 control's **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 SocketTools controls have 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 web server uses port 80 to accept connections, while an FTP server uses port 21. In most cases, it is not necessary to explicitly specify a port number because the SocketTools components will automatically select the correct port number for the protocol being used. However, in some cases servers are configured to use non-standard port numbers. 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. Note that the File Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP) controls do provide a **URL** property which can be assigned a URL string and the control will automatically parse the URL and set the corresponding HostName and RemotePort properties to their correct values. Refer to the Technical Reference for more information.

Asynchronous Sessions

The SocketTools controls have 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, the control 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, the control 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 control 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 control 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 nError As Long

nError = FtpClient1.Connect("ftp.microsoft.com")
If nError = 0 Then
    nError = FtpClient1.Login()
End If

If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

In this example, the FTP control is being used to establish a connection to ftp.microsoft.com. If no error is returned, then the program attempts to login the user. If an error does occur, a message box is displayed and the subroutine is exited. This code is fairly straight-forward and would work as expected with a synchronous connection where the **Blocking** property is set to True. However, if the control was set to use an asynchronous connection 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

Login method would need to be moved out of that code block and into the OnConnect event handler.

When the control 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 in most cases 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 the control in asynchronous mode, we recommend that you always 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. For languages such as Visual C++ and Visual Basic.NET which support multithreading, 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 control to block during execution, regardless of what mode the control is using. An example of this is the **GetFile** method in the FTP and HTTP controls, which downloads a file from the server to the local system. To use the control in asynchronous mode, you are limited to using the lower-level methods such as **OpenFile**.

Quick Start Control Overview

This section of the Quick Start Guide provides a general overview of the SocketTools ActiveX controls used for the most common tasks, divided into groups based on their functionality. For a complete list and detailed descriptions of all of the controls included in the product, refer to the Control Overview section of the Developer's Guide.

File Transfer

If your program needs to transfer files between a local computer system and a server, SocketTools includes components which implement the File Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP). Both protocols can be used to upload and download files, and FTP also supports various file management functions. Which protocol is used largely depends on the specific needs of the application and the type of server that the program is connecting to. A general purpose file transfer program would most likely use both controls and allow the user to select which protocol to use.

Control	Description
FtpClient	The File Transfer Protocol control enables an application to upload and download files, as well as perform various file management functions on the server. For example, the control can be used to list the files in a directory, delete and rename files, etc.
HttpClient	The Hypertext Transfer Protocol control enables an application to upload and download files, as well as interact with web-based applications. The file management capabilities are somewhat limited compared to FTP, however the protocol is not as complex and has fewer compatibility issues with certain network configurations.

Web Services

Applications which need to access resources on a web server and interact with web-based applications can use the Hypertext Transfer Protocol (HTTP) control. In the context of web services, the control can be used to access resources on a web server, execute scripts and other applications, as well as perform various management functions using WebDAV, a protocol extension for distributed authoring.

Control	Description
HttpClient	The Hypertext Transfer Protocol control enables an application to upload and download files, as well as interact with web-based applications. The control can be used to post data to scripts which are executed on the server and return the output of those scripts to the client application.

email Services

SocketTools includes several controls which can be used to create applications that send and receive email messages. Applications can compose, edit and store messages on the local system, retrieve messages from a mail server and send messages to one or more recipients. The SocketTools controls support features such as the ability to compose messages with styled (HTML) text, file attachments, relay server authentication and delivery status notification. Programs which only wish to process messages sent to a user would typically use the MIME control and either the IMAP4 or POP3 controls. Programs which only wish to send messages would typically use the MIME control and the SMTP control. A full featured mail client would use all of the following components.

Control	Description
	The Multipurpose Internet Mail Extensions (MIME) standard defines the structure and format which is used by email messages. This control enables you to create MIME

MailMessage	compliant messages easily, as well as parse existing messages, edit them and store them on the local system. The control supports complex multipart attachments, including messages with one or more file attachments and messages with alternative content such as styled HTML text.
ImapClient	The Internet Message Access Protocol (IMAP4) control can be used to manage email messages on a mail server. Using this control, you can list and retrieve messages, search for specific messages, manage multiple mailboxes, retrieve portions of a message and perform other advanced functions.
PopClient	The Post Office Protocol (POP3) control can be used to list the messages on a mail server and download them to the local system. Unlike the IMAP4 protocol, which is designed to manage messages on the server, the POP3 protocol is used primarily to retrieve messages, store them locally and then delete them from the server. POP3 is a simpler protocol with less functionality than IMAP4, however it more widely supported.
SmtpClient	The Simple Mail Transfer Protocol (SMTP) control is used to submit a message for delivery to one or more recipients. The control can be used to either send the message directly to the recipient, or messages can be routed through a relay server which is responsible for forwarding the message. Both standard SMTP and extended ESMTP sessions are supported, along with advanced options such as authentication and delivery status notification.

Terminal Services

Applications which need to execute commands on a server or establish a terminal session can use the SocketTools Telnet and Remote Shell (RSH) controls. The program can connect to the server and interact with the server in the same way that a user can with a character based terminal. In addition, SocketTools includes a terminal emulation control which can be used to emulate an ANSI console or a DEC VT-220 terminal. This can be used to either provide the user with a traditional virtual terminal interface, or the program can read data at specific rows and columns and effectively provide a graphical interface for a legacy character-based application running on the server.

Control	Description
TelnetClient	The Telnet protocol control enables the application to establish a standard, interactive terminal session with a server. This approach is similar to how character-based terminals were connected to systems and users would login to the mainframe or minicomputer. For legacy applications that run on a UNIX server, this control can be used to connect to the server, login and interact with the server just as a user would sitting at a terminal. You can either choose to display the terminal session to the user, or you can have your application present a graphical interface to the user and interact with the terminal session in the background.
RshClient	The Remote Shell (RSH) control actually implements three related protocols in a single component. The rshell and rexec protocols are used to execute a command on the server and the output from that command is returned to the client. The difference between the two protocols has to do with how authentication is handled. The rexec protocol uses a password to authenticate a user session, while rshell uses host equivalence. More information about these protocols is available in the Technical Reference. This control also implements the rlogin protocol, which is similar to the Telnet protocol in that it provides an interactive terminal session.
Terminal	The Terminal control emulates a standard character-based terminal, either as an ANSI console, DEC VT-100 or DEC-VT220 terminal. The emulator supports all of the standard ANSI and DEC escape sequences, including support for colors and line drawing. Your

program has full control over the functionality of the control, including the color mapping, the escape sequences that special keys (such as the function keys) send and whether the user can do things such as select and copy text from the virtual display.

Common Properties

Many of the SocketTools components share a common set of properties, each with the same general functionality. This approach makes it easier to understand the interface and reduces the overall learning curve. However, it is important to note that some common properties may affect the operation of the controls in different ways. Although this guide can provide a general overview of those properties and how they are used, it is recommended that you also review the Technical Reference material for the control that you are using in your application.

Property Name	Description
AutoResolve	Determines if host names and IP addresses are automatically resolved
Blocking	Gets and sets the blocking state of the control
HostAddress	Gets and sets the IP address of the server
HostName	Gets and sets the name of the server
IsBlocked	Return if the control is blocked performing an operation
IsConnected	Determine if the control is connected to a server
IsReadable	Return if data can be read from the server without blocking
IsWritable	Return if data can be sent to the server without blocking
LastError	Gets and sets the last error that occurred on the control
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
Password	Gets and sets the password for the current user
RemotePort	Gets and sets the port number for a remote connection
ResultCode	Return the result code of the previous action
ResultString	Return a string describing the results of the previous action
Secure	Set or return if a connection to the server is secure
State	Return the current state of the control
ThrowError	Enable or disable error handling by the container of the control
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
UserName	Gets and sets the current user name
Version	Return the current version of the object

AutoResolve

The **AutoResolve** property controls how host names are resolved by the control whenever the

HostName or HostAddress properties are set. By default, the property is set to False, which means that the control does not attempt to resolve the host name until a connection attempt is made. If the property is set to True, then the control will immediately attempt to resolve the host name into an IP address. Note that this can cause the control 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 control operates in blocking (synchronous) mode or non-blocking (asynchronous) mode. In blocking mode, the control 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 a control 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 server, 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.

In general, 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 control to establish connections to different servers, it is preferable to create a multithreaded application rather than attempt to use multiple instances of the control in a single thread.

HostAddress

The **HostAddress** property is used to specify the IP address of a server 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 control 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 server 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 control 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 control is currently performing a blocking operation. This can be used in conjunction with the **State** property to determine if the control can be used to issue a command to the server or perform some other operation. When the **IsBlocked** property returns False and the **State** property returns a value of zero or one, the control 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 stErrorOperationInProgress 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 control 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 control can successfully write data using the **Write** method. If the property returns False, then the control's internal buffers are full and cannot accept any more data until the server 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 control 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 control 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.

Password

The **Password** property is used to specify the password used to authenticate the client session with the server. This property is only used by those controls which support authentication. Setting this property to an empty string will clear the current password being used. This property should be used in conjunction with the **UserName** property.

RemotePort

The **RemotePort** property is used to specify the port number used to establish a connection with the server. A value of zero specifies that the default port number for the protocol should be used. For example, if the property is set to zero with the FTP control, then the control will use port 21 by default. 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.

ResultCode

The **ResultCode** property returns the last numeric result code sent by the server in response to a command. Result codes are used to determine the status of a command issued by a server, typically indicating success, failure or that the client must provide additional information. It is important to note that different protocols use result codes in different ways. Refer to the Technical Reference for more information about how result codes are returned by a specific control. To obtain a description of the result code, use the **ResultString** property.

ResultString

The **ResultString** property returns a description of the last result code sent by the server in response to a command. The values of these strings are completely dependent on the server implementation and can vary from server to server. An application should never depend on a server returning a specific description of a command result and instead should rely on the **ResultCode** property. The result string is primarily used to provide additional information to the user or for debugging purposes.

Secure

The **Secure** property determines if the control 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 control will attempt to establish a secure connection using the Transport Layer Security (TLS) protocol.

State

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

ThrowError

The **ThrowError** property is used to determine how errors are reported by the control when calling a method. The default value is False, which specifies that errors should be returned as values from the method call and the control 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 control 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. Note that the Internet Control Message Protocol (ICMP) control is an exception in that the **Timeout** property specifies a value in milliseconds, not 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 control. When the property is set to True, the control 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 cstrcv11.dll with your application. If this library cannot be loaded, 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 of zero specifies that all of the networking function calls should be logged, along with their arguments and return values. The following values are used:

Value	Description
All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.	
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.	
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.	
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 server.	

UserName

The **UserName** property is used to specify the username used to authenticate the client session with the server. This property is only used by those controls which support authentication. Setting this property to an empty string will clear the current username. This property should be used in conjunction with the **Password** property.

Version

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

Common Methods

Many of the SocketTools components share a common set of methods, each with the same general functionality. It is important to note that although methods may share the same name, the number and type of arguments may vary from control to control. Be sure to review the Technical Reference documentation for the specific control that you are using.

Method Name	Description
Cancel	Cancels the current blocking network operation
Command	Send a custom command to the server
Connect	Establish a connection with a server
Disconnect	Terminate the connection with a server
Initialize	Initialize the control and validate the runtime license key
Read	Return data read from the server
Reset	Reset the internal state of the control
Uninitialize	Uninitialize the control and release any system resources allocated for the session
Write	Write data to the server

All methods expect that the arguments passed to them will be variants, and the argument will be converted to the appropriate type by the method. This allows the controls to be easily used by weakly typed languages, and is generally transparent to the caller in languages such as Visual Basic. However, in languages such as Visual C++, you should either use VARIANT types or related classes such as CComVariant when passing arguments to a method. In most cases, the arguments should be passed by value to the method, however there are exceptions where a method returns data to the caller in one or more arguments. In this case, the caller must pass the argument by reference. Again, languages such as Visual Basic will handle this for you transparently. However, some languages such as FoxPro or PowerBuilder require that you use special syntax when passing an argument by reference. Refer to the documentation for your programming language if you have any questions on how to pass an argument by reference to a method.

Most methods will return a value of zero if they are successful, or an error code value if the method fails for some reason. This error code will match the value of the **LastError** property, and a 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 for each control.

Cancel

The **Cancel** method cancels the current blocking operation being performed by the control. 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 stErrorOperationCanceled. 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.

Command

The **Command** method is used to send custom commands to the server, specific to the protocol being used. This provides the program with a very low level of access to the application protocol.

Typically it is used to take advantage of non-standard extensions to the protocol or server-specific commands. After calling the **Command** method, the program should check the value of the ResultCode and ResultString properties to determine if the command was successful or an error occurred.

Connect

The **Connect** method is used to establish a connection with a server, and is typically one of the first methods called by the program. The method accepts one or more optional arguments, such as the host name or IP address of the server, the port number, and in some cases a user name and password. If no arguments are specified, then the method will use the values of the HostName or HostAddress, RemotePort, UserName and Password properties as the default. If the control is in blocking mode, then the method will return after the connection has been established, or after the timeout period has been exceeded. If the control is in non-blocking mode, the method will return immediately and the application must wait for either the **OnConnect** or **OnError** events to fire.

Disconnect

The **Disconnect** method terminates the current connection and releases some of the resources allocated by the control 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 control, loading the appropriate networking libraries, validating the runtime license key and performing other internal initialization functions. If the control is placed on a form or dialog, then it is not normally required that a program call this method because the container (form) will automatically initialize the control for you. However, if the control is created dynamically using CreateObject or created using a reference, then 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 the control 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 Control Initialization section of the Developer's Guide.

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 and the command issued to the server. The first argument passed to the method should be a string or byte array which will contain the data that has been read. The second argument should be an integer which specifies the amount of data to read, in bytes. The **Read** method is typically only used in conjunction with those methods which provide lower-level access to the application protocol.

The **Read** method is different from most other methods in two important ways. Instead of returning zero or an error code, the method returns the number of bytes read. If an error occurs, then the method will return -1. 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, 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. Consult the documentation for your programming language if you have any questions about how to do this.

Reset

The **Reset** method will reset the internal state of the control to its defaults, terminating any connection to the server and releasing resources allocated for the client session. This method should only be used when the program needs to effectively abort any connection 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 library and release those resources which have been allocated by the control. In most cases, it is not necessary to explicitly uninitialize the control because this is handled automatically when the control is unloaded from the form or the application terminates. For each call to the **Initialize** method, there should be a matching call made to the **Uninitialize** method when the control is no longer being used.

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 typically only used in conjunction with those methods which provide lower-level access to the application protocol.

The **Write** method is different from most other methods because instead of returning zero or an error code, 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.

Common Events

Many of the SocketTools components share a common set of events, each with the same general functionality. It is important to note that although events may share the same name, the number and type of arguments may vary from control to control. Be sure to review the Technical Reference documentation for the specific control that you are using.

Event Name	Description
OnCancel	This event is generated when a blocking operation is canceled
OnCommand	This event is generated when the server processes a command issued by the client
OnConnect	This event is generated when a connection is established
OnDisconnect	This event is generated when a connection is terminated
OnError	This event is generated when a control error occurs
OnProgress	This event is generated during data transfer
OnRead	This event is generated when data is available to be read
OnTimeout	This event is generated when a blocking operation times out
OnWrite	This event is generated when data can be written to the server

The events generated by the SocketTools controls 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 control 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 events have their arguments passed by value as variants. For languages such as Visual Basic, this does not require any special consideration when implementing an event handler. For other languages, you may need to convert the variant into the appropriate data type for your application. For example, in Visual C++ this can be done using the standard macros included in oleauto.h or using a class such as CComVariant. For more information about how to implement an event handler in C++, refer to to the section on Control Event Handling in the Developer's Guide.

When developing your event handlers, it is important to remember that the 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.

OnCancel

The **OnCancel** event is generated whenever a blocking operation has been canceled using the **Cancel** method. When this event fires, the component is about to return control to your application, and the blocked method will return with the error stErrorOperationCanceled. 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 function.

OnCommand

The **OnCommand** event is generated whenever the server has processed a command issued by the client. The event handler is passed two arguments, the numeric result code and a string describing the result of the command. These values correspond to the ResultCode and ResultString properties. This event is typically used by applications to record the responses from a server, either as information for the user or for debugging purposes.

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 server. This event is only generated when the **Blocking** property is set to False. If the control 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 server 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.

OnError

The **OnError** event occurs whenever an error is reported by the control. The event handler is passed two arguments, the numeric error code and a 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, providing information to the application about the status of the transaction. For example, this event is called periodically during a file transfer so that the program knows how much of the file has been uploaded or downloaded. 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. This event is only generated when the **Blocking** property is set to False and a lower-level method is called which requires the application to read data directly from the server. 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 some of the 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 component is about to return control to your application, and the blocked method will return with the error stErrorOperationTimeout. 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 function.

OnWrite

The **OnWrite** event is a networking event which occurs whenever the server is able to receive more data from the application. This event is only generated when the **Blocking** property is set to False and a previous call to the **Write** method failed with the error stErrorOperationWouldBlock.

File Transfers

SocketTools includes two components which are primarily used for uploading and downloading files. The File Transfer Protocol (FTP) control is used to transfer files between the local system and an FTP server on a server. This control also enables an application to list directories and search for files, perform remote file management tasks such as renaming and deleting files, as well as a number of other functions. The Hypertext Transfer Protocol (HTTP) control can also be used to transfer files, and the properties, methods and events for the two controls are very similar.

A common question asked by most developers is which is the best protocol to use if you want to transfer files. This depends on a number of factors, including the type of server being connected to and what the program needs to do. As a general rule, FTP is a more complex protocol which offers more features in terms of the ability to manage files on a server. For example, you can use FTP to list all of the files in a directory on the server, search for those files which were modified after a certain date and download them. HTTP doesn't provide those kinds of facilities. However, FTP can present problems when it is used behind a firewall or a router which uses Network Address Translation (NAT) and applications which use FTP need to give users the ability to configure various options, such as whether or not passive-mode file transfers are used.

On the other hand, HTTP offers a much simpler protocol and typically does not have the same kinds of problems when used from behind a firewall or NAT router. However, it does not have many of the more advanced file management features that FTP supports, and the ability to upload or delete files usually requires that the administrator of the web server make specific changes to allow this.

Here are some general guidelines you can follow to determine which is the best protocol to use in a given situation:

- If your program only downloads files from the server to the local system, use HTTP whenever possible.
- If your program only uploads files from the local system to the server, use FTP whenever possible. If you must use HTTP, make sure that your web administrator has enabled the use of the PUT command.
- If your program frequently downloads files from the server and occasionally uploads files, use HTTP whenever possible and make sure that your web administrator has enabled the use of the PUT command.
- If your program must perform file management functions such as deleting and renaming files, creating directories or searching for files which meet a specific criteria, use FTP whenever possible.
- If you want to use a secure, encrypted connection to protect the contents of the data being transferred, use HTTP whenever possible. Although there are extensions to FTP which support secure connections, those servers are not as common as secure web servers.
- For broadest compatibility, use both the FTP and HTTP controls and allow your user to select which protocol to use and the various options that are available.

If you decide to use the FTP control, make sure that you set the Passive property to True. This will force the control to only establish outbound connections to the server and ensures a broader range of compatibility with various firewall and router configurations.

Establishing a Connection

In order to transfer files or use the various file management methods, it is necessary to establish a connection to the server. This is done by calling the Connect method and checking the return value to make sure that the connection was successful. To establish that connection, certain information is required. This includes:

- The host name or IP address of the server that you wish to connect to. This information may be specified by either setting the **HostName** property, or by passing the host name as an argument to the **Connect** method. For example, ftp.microsoft.com is a valid host name.

- The port number of the server that you're connecting to. This can be specified by setting the **RemotePort** property or passing the port number as an argument to the **Connect** method. In most cases, the default port number should be used and it is not necessary to specify the port number in your application. However, for servers that are configured to use non-standard port numbers, that information must be provided. Setting the **RemotePort** property to a value of zero specifies that the default port number appropriate for that protocol should be used.

- If a secure, encrypted connection is required, then the **Secure** property should be set to True. A secure server is one that supports the TLS (Transport Layer Security) protocol or the SSH (Secure Shell) protocol.

In addition, FTP connections almost always require a user name and password. If neither is specified, the control will attempt an anonymous login. Not all FTP servers support anonymous logins and an error may be returned indicating that the username or password is invalid. In rarer circumstances, an account name may also be required. HTTP connections rarely require a username and password for downloads, although that information may be required to upload files to the server.

Either FTP or HTTP servers may be accessible only through a proxy server in certain circumstances. In those circumstances, a set of properties related to proxies must be specified before connecting. Please see the Technical Reference for further details.

File Transfer Protocol Connections

To establish a connection using the FTP control, you should call the **Connect** method and check the return value to ensure that the connection was successful. This can be done one of several ways, and which approach you use largely depends on personal preference and the structure of your program.

Method Arguments

You can call the **Connect** method with a number of different arguments, all of which are optional. The most common use would look like this:

```
Dim nError As Long

nError = FtpClient1.Connect(strHostName, ftpPortDefault, strUserName, strPassword)
If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

In this example, the strHostName string variable contains the name of the server to connect to, the strUserName variable contains the username and the strPassword variable contains the password used to authenticate that user. If the function returns a value other than zero, this indicates an error and a message box is used to display the error to the user.

Property Values

Another method is to set the host name, user name and password using the control's properties and then calling the **Connect** method without any arguments. Here is an example of how that could be done:

```
Dim nError As Long

FtpClient1.HostName = "ftp.microsoft.com"
FtpClient1.UserName = "anonymous"
FtpClient1.Password = "user@domain"

nError = FtpClient1.Connect()
If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

Note that the **RemotePort** property is not specified, which means that the default port number should be used. As with the previous example, if the connection fails then the method will return an error code and a message box is displayed to the user.

Uniform Resource Locators

A third approach is to specify a URL by setting the **URL** property and then calling the **Connect** method without any arguments. By setting the URL property, the control will automatically update the various related properties such as **HostName** and **RemotePort**. Here is an example of how this could be done:

```
Dim nError As Long

FtpClient1.URL = "ftp://ftp.microsoft.com/"

nError = FtpClient1.Connect()
If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

If you prefer to use URLs, this approach provides the simplest means of establishing the connection. It should be noted that if the URL property is assigned an invalid value, then the control will throw an exception. If you are using URLs provided by a user, it is strongly recommended that you implement an error handler, otherwise your program may terminate if the user specifies an incorrect or incomplete URL.

Hypertext Transfer Protocol Connections

To establish a connection using the HTTP control, you should call the **Connect** method and check the return value to ensure that the connection was successful. This can be done one of several ways, and which approach you use largely depends on personal preference and the structure of your program.

Method Arguments

You can call the **Connect** method with a number of different arguments, all of which are optional. The most common use would look like this:

```
  Dim nError As Long

  nError = HttpClient1.Connect(strHostName)
  If nError > 0 Then
      MsgBox HttpClient1.LastErrorString, vbExclamation
      Exit Sub
  End If
```

In this example, the strHostName string variable contains the name of the server to connect to. If the method returns a value other than zero, this indicates an error and a message box is used to display the error to the user. Note that if the resource requires authentication, then a username and password can also be passed to the method.

Property Values

Another method is to set the host name using the control's **HostName** property and then calling the **Connect** method without any arguments. Here is an example of how that could be done:

```
Dim nError As Long

HttpClient1.HostName = "www.microsoft.com"

nError = HttpClient1.Connect()
If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

Note that the **RemotePort** property is not specified, which means that the default port number should be used. If authentication was required, the **UserName** and **Password** properties could be set as well. As with the previous example, if the connection fails then the method will return an error code and a message box is displayed to the user.

Uniform Resource Locators

A third approach is to specify a URL by setting the **URL** property and then calling the **Connect** method without any arguments. By setting the URL property, the control will automatically update the various related properties such as **HostName** and **RemotePort**. Here is an example of how this could be done:

```
  Dim nError As Long

  HttpClient1.URL = "http://www.microsoft.com/"

  nError = HttpClient1.Connect()
  If nError > 0 Then
      MsgBox HttpClient1.LastErrorString, vbExclamation
      Exit Sub
  End If
```

If you prefer to use URLs, this approach provides the simplest means of establishing the connection. It should

be noted that if the URL property is assigned an invalid value, then the control will throw an exception. If you are using URLs provided by a user, it is strongly recommended that you implement an error handler, otherwise your program may terminate if the user specifies an incorrect or incomplete URL.

Downloading Files

The **GetFile** method is used to download files using either the FTP or HTTP controls. In its simplest form, the method accepts two arguments, the name of the local file to create or overwrite and the name of the file on the server to download. For example, consider a form with three TextBox controls and a CommandButton control:

```
Private Sub Command1_Click()
    Dim strHostName As String
    Dim strLocalFile As String
    Dim strRemoteFile As String
    Dim nError As Long

    strHostName = Trim(Text1.Text)
    strLocalFile = Trim(Text2.Text)
    strRemoteFile = Trim(Text3.Text)

    ' Establish a connection to the server and display any
    ' errors to the user
    nError = FtpClient1.Connect(strHostName)
    If nError Then
        MsgBox FtpClient1.LastErrorString, vbExclamation
        Exit Sub
    End If

    ' Download the file to the local system
    nError = FtpClient1.GetFile(strLocalFile, strRemoteFile)
    If nError Then
        MsgBox FtpClient1.LastErrorString, vbExclamation
        FtpClient1.Disconnect
        Exit Sub
    End If

    ' Disconnect from the server
    FtpClient1.Disconnect
End Sub
```

In this example, there is no user name or password specified so the FTP server would need to accept anonymous logins. This same code would work with the HTTP control to download a file from a web server. To provide some feedback to the user as to the status of the transfer, a ProgressBar control could be added to the form. To update the progress bar, a handler for the **OnProgress** event would also be added:

```
Private Sub FtpClient1_OnProgress(ByVal FileName As Variant, _
                                  ByVal FileSize As Variant, _
                                  ByVal BytesCopied As Variant, _
                                  ByVal Percent As Variant)
    ProgressBar1.Value = Percent
End Sub
```

Simply setting the **Value** property of the ProgressBar control to the **Percent** argument in the event handler will update the progress bar during the file transfer process. As with the previous code, this can also be done using the HTTP control in the same way.

For a second example, let's use the HTTP control to download an HTML page from a website and store it on the local system. The form has two TextBox controls and a CommandButton control. The first TextBox control will contain the URL of the page to download, and the second control will specify the local name on the server.

```
Private Sub Command1_Click()
    Dim strHostName As String
```

```
    Dim strLocalFile As String
    Dim strRemoteFile As String
    Dim nError As Long

    ' If the user enters an invalid URL, setting the URL property will
    ' throw an exception, so that needs to be handled here
    On Error Resume Next: Err.Clear
    HttpClient1.URL = Trim(Text1.Text)

    If Err.Number Then
        MsgBox "An invalid URL has been specified", vbExclamation
        Exit Sub
    End If

    On Error GoTo 0
    strLocalFile = Trim(Text2.Text)

    ' Establish a connection to the server and display any
    ' errors to the user
    nError = HttpClient1.Connect()
    If nError Then
        MsgBox HttpClient1.LastErrorString, vbExclamation
        Exit Sub
    End If

    ' Download the file to the local system
    nError = HttpClient1.GetFile(strLocalFile, HttpClient1.Resource)
    If nError Then
        MsgBox HttpClient1.LastErrorString, vbExclamation
        HttpClient1.Disconnect
        Exit Sub
    End If

    ' Disconnect from the server
    HttpClient1.Disconnect
End Sub
```

You'll notice that much of the code is very similar, with the exception for the error handling code used when the **URL** property is set, and the use of the **Resource** property. When the **URL** property is set, the control will automatically parse the **URL** and update the related properties such as **HostName** and **RemotePort**. The path and file name portion of the URL is assigned to the **Resource** property, which can then be used in conjunction with the methods such as **GetFile** and **PutFile**.

Uploading Files

The **PutFile** method is used to upload files using either the FTP or HTTP controls. In its simplest form, the method accepts two arguments, the name of the local file to upload and the name of the file that will be created or overwritten on the server. For example, consider a form which has five TextBox controls and a CommandButton control:

```
Private Sub Command1_Click()
    Dim strLocalFile As String
    Dim strRemoteFile As String
    Dim nError As Long

    FtpClient1.HostName = Trim(Text1.Text)
    FtpClient1.UserName = Trim(Text2.Text)
    FtpClient1.Password = Trim(Text3.Text)
    strLocalFile = Trim(Text4.Text)
    strRemoteFile = Trim(Text5.Text)

    ' Establish a connection to the server and display any
    ' errors to the user
    nError = FtpClient1.Connect()
    If nError Then
        MsgBox FtpClient1.LastErrorString, vbExclamation
        Exit Sub
    End If

    ' Download the file to the local system
    nError = FtpClient1.PutFile(strLocalFile, strRemoteFile)
    If nError Then
        MsgBox FtpClient1.LastErrorString, vbExclamation
        FtpClient1.Disconnect
        Exit Sub
    End If

    ' Disconnect from the server
    FtpClient1.Disconnect
End Sub
```

In this example, the user specifies the name of a server, a username and a password. In most cases, servers only permit authenticated users to upload files, so this information is usually required. This same code would work with the HTTP control to upload a file to a web server. To provide some feedback to the user as to the status of the transfer, a ProgressBar control could be added to the form. To update the progress bar, a handler for the **OnProgress** event would also be added:

```
Private Sub FtpClient1_OnProgress(ByVal FileName As Variant, _
                                  ByVal FileSize As Variant, _
                                  ByVal BytesCopied As Variant, _
                                  ByVal Percent As Variant)
    ProgressBar1.Value = Percent
End Sub
```

Simply setting the **Value** property of the ProgressBar control to the **Percent** argument in the event handler will update the progress bar during the file transfer process. As with the previous code, this can also be done using the HTTP control in the same way.

Listing Files

In addition to uploading and downloading files, the FTP control can also be used to list the files that are in a specific directory on the server. This is done by using three methods: **OpenDirectory**, **ReadDirectory** and **CloseDirectory**. For example, consider a form with two TextBox controls, a ListBox control and a CommandButton control. The first TextBox control is used to specify the host name of the server and the second control specifies the directory. The ListBox control is populated with the name of the files in that directory.

```
Private Sub Command1_Click()
    Dim strHostName As String
    Dim strDirectory As String
    Dim strFileName As String
    Dim nError As Long

    strHostName = Trim(Text1.Text)
    strDirectory = Trim(Text2.Text)

    ' Establish a connection to the server and display any
    ' errors to the user
    nError = FtpClient1.Connect(strHostName)
    If nError Then
        MsgBox FtpClient1.LastErrorString, vbExclamation
        Exit Sub
    End If

    ' Open the directory on the server to begin the
    ' process of reading the contents
    nError = FtpClient1.OpenDirectory(strDirectory)
    If nError Then
        MsgBox FtpClient1.LastErrorString, vbExclamation
        FtpClient1.Disconnect
        Exit Sub
    End If

    ' Read each file name until the end of the directory
    ' listing is reached
    Do
        nError = FtpClient1.ReadDirectory(strFileName)
        If nError Then Exit Do
        List1.AddItem strFileName
    Loop

    If nError <> stErrorEndOfDirectory Then
        MsgBox FtpClient1.LastErrorString, vbExclamation
    End If

    ' Close the directory and disconnect from the server
    FtpClient1.CloseDirectory
    FtpClient1.Disconnect
End Sub
```

In this example, the **ReadDirectory** method is used to return only the name of the file, however it can return additional information such as the file size, date it was last modified, its access permissions and whether or not its a regular file or a subdirectory. For more information, refer to the Technical Reference documentation for this method.

File Management

The FTP control includes a number of methods which can be used to manage files and directories on a server. In most cases, it is required that the client authenticate itself with a username and password because few servers support these functions when the client is logged in anonymously. It is also required that the user have the appropriate access rights on the server to perform the requested function. For example, the user must have the right to modify a directory in order to delete a file from that directory. If a method returns an error indicating that access was denied, verify with the server administrator that you have the correct access rights to that file or directory.

Renaming Files

The FTP control supports the ability to rename files using the **RenameFile** method. It take two arguments, the name of the file on the server and the new name that you want to change it to:

```
nError = FtpClient1.RenameFile(strOldName, strNewName)
```

If the file is renamed successfully, the method will return zero, otherwise it will return an error code. Note that you can also use this method to move files between directories and rename directories as well.

Deleting Files

Both the FTP and the HTTP control supports the ability to delete files using the **DeleteFile** method. The method accepts a single string argument, the name of the file to delete:

```
nError = FtpClient1.DeleteFile(strFileName)
```

If the file is deleted successfully, the method will return zero, otherwise it will return an error code. Deleting a file is a permanent action and there is no provision for restoring a previously deleted file. This method cannot be used to remove a directory. If this method is used with the HTTP control, it requires that the server be configured to allow file deletion. Most web servers do not permit this by default.

Creating Directories

The FTP control can be used to create new directories using the **MakeDirectory** method. This method accepts a single string argument which specifies the name of the directory to create:

```
nError = FtpClient1.MakeDirectory(strNewDirectory)
```

If the directory has been created successfully, the method will return zero, otherwise it will return an error code. Most servers will not create multiple subdirectories in a single operation. For example, if you specify the name "/office/projects/documents" an error will usually be returned if the directory "/office/projects" does not already exist.

It is also important to keep in mind that the file naming conventions depend on the server operating system. For example, a server running on a UNIX system uses case-sensitive file names, which means that the directory "/Office/Projects" is different than "/office/projects". On the other hand, those would refer to the same directory on a Windows server. If this distinction is important to your program, you can determine the type of server that you're connected to by checking the value of the System property.

Deleting Directories

The FTP control can be used to delete directories using the **RemoveDirectory** method. This method accepts a single string argument which specifies the name of the directory to delete:

```
nError = FtpClient1.RemoveDirectory(strDirectory)
```

If the directory has been deleted successfully, the method will return zero, otherwise it will return an error code. Deleting a directory is a permanent action and there is no provision for restoring a previously deleted directory. If the directory is not empty (in other words, contains one or more files or subdirectories) then the method will fail.

Checking Modification Times

If you need to determine the time that a file was last modified, both the FTP and HTTP controls provide a method called **GetFileTime**. This method has two arguments, the name of the file to check, and a string variable passed by reference which will contain the file date and time when the method returns. For example:

```
nError = FtpClient1.GetFileTime(strFileName, strFileDate)
If nError = 0 Then
    MsgBox "The file was modified on " & strFileDate
End If
```

The date string is formatted according to the system locale and the value of the **Localize** property determines if the date is adjusted for the local timezone. If the file does not exist, or the server does not support the command used to obtain the modification time for the file, an error will be returned.

Checking Permissions

The FTP control can be used to determine the access permissions that have been specified for a given file using the **GetFilePermissions** method. This method accepts two arguments, the name of the file to check and an integer passed by reference which will contain the file permissions when the method returns. For example:

```
nError = FtpClient1.GetFilePermissions(strFileName, nFilePerms)
If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

If (nFilePerms And ftpPermOwnerRead) <> 0 Then
    MsgBox "The file " & strFileName & " can be read by the owner"
End If
```

The integer value returned is one or more bit flag values which contains information about the access rights for the file. For more information, refer to the table in the Technical Reference documentation. If the file does not exist, or the server does not support the command used to obtain the file permissions, an error will be returned.

Changing Permissions

In addition to checking the access rights for a file, you can also change those rights using the **SetFilePermissions** method in the FTP control. The following example demonstrates how to change the permissions so that only the owner can read and write to the file:

```
nFilePerms = ftpPermOwnerRead Or ftpPermOwnerWrite
nError = FtpClient1.SetFilePermissions(strFileName, nFilePerms)
If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

For more information about the permission flags that can be used, refer to the Technical Reference documentation. If the file does not exist, or the server does not support the command used to change the file permissions, an error will be returned.

Web Services

The SocketTools Hypertext Transfer Protocol (HTTP) control can be used to interact with a web server, requesting resources, executing scripts and submitting form data for processing. This section of the Quick Start Guide will cover how to use the control to establish connections to a web server, request resources and post form data to a script.

In addition the HTTP control also supports protocol extensions such as WebDAV for distributed authoring. It can be used to automate certain processes and enable your application to interact directly with a program running on the server without having to use a browser interface.

Establishing a Connection

In order to access a resource on a web server, it is first necessary to establish a connection. This is done by calling the **Connect** method and checking the return value to make sure that the connection was successful. To establish that connection, certain information is required. This includes:

- The host name or IP address of the server that you wish to connect to. This information may be specified by either setting the **HostName** property, or by passing the host name as an argument to the **Connect** method. For example, www.microsoft.com is a valid host name.

- The port number of the server that you're connecting to. This can be specified by setting the **RemotePort** property or passing the port number as an argument to the **Connect** method. In most cases, the default port number should be used and it is not necessary to specify the port number in your application. However, for servers that are configured to use non-standard port numbers, that information must be provided. Setting the **RemotePort** property to a value of zero specifies that the default port number should be used.

- If a secure, encrypted connection is required, then the **Secure** property should be set to True. A secure server is one that supports the TLS (Transport Layer Security) protocol.

HTTP servers may only be accessible through a proxy server in certain circumstances. In that case, a set of properties related to proxies must be specified before connecting. Please see the Technical Reference for further details about the **ProxyHost** and **ProxyPort** properties, as well as other information pertaining to connecting through a proxy server.

To establish a connection using the HTTP control, you should call the **Connect** method and check the return value to ensure that the connection was successful. This can be done one of several ways, and which approach you use largely depends on personal preference and the structure of your program.

Method Arguments

You can call the **Connect** method with a number of different arguments, all of which are optional. The most common use would look like this:

```
Dim nError As Long

nError = HttpClient1.Connect(strHostName)
If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

In this example, the strHostName string variable contains the name of the server to connect to. If the function returns a value other than zero, this indicates an error and a message box is used to display the error to the user. Note that if the resource requires authentication, then a username and password can also be passed to the method.

Property Values

Another method is to set the host name using the control's **HostName** property and then calling the **Connect** method without any arguments. Here is an example of how that could be done:

```
Dim nError As Long

HttpClient1.HostName = "www.microsoft.com"

nError = HttpClient1.Connect()
```

```
If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

Note that the **RemotePort** property is not specified, which means that the default port number should be used. If authentication was required, the **UserName** and **Password** properties could be set as well. As with the previous example, if the connection fails then the method will return an error code and a message box is displayed to the user.

Uniform Resource Locators

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

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

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

A third approach to establishing a connection is to specify a URL by setting the **URL** property and then calling the **Connect** method without any arguments. By setting the **URL** property, the control will automatically update the various related properties such as **HostName** and **RemotePort**. Here is an example of how this could be done:

```
Dim nError As Long

HttpClient1.URL = "http://www.microsoft.com/"

nError = HttpClient1.Connect()
If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

If you prefer to use URLs, this approach provides the simplest means of establishing the connection. It should be noted that if the **URL** property is assigned an invalid value, then the control will throw an exception. If you are using URLs provided by a user, it is strongly recommended that you implement an error handler, otherwise your program may terminate if the user specifies an incorrect or incomplete URL.

Web Server Authentication

In some cases, it is required that the client authenticate itself to the web server prior to requesting a resource. This can be done in one of two ways, either by setting the UserName and Password properties or by providing those values to the **Connect** method when it is called. If you attempt to access a resource that requires authentication, you'll get the error stErrorCommandNotAuthorized.

Method Arguments

You can call the **Connect** method and specify the user name and password as arguments. The most common use would look like this:

```
Dim strUserName As String
Dim strPassword As String
Dim nError As Long

nError = HttpClient1.Connect(strHostName, , strUserName, strPassword)
If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

In this example, the strHostName string variable contains the name of the server to connect to and strUserName and strPassword provide the credentials to authenticate the client session. Note that we omit the argument that specifies a remote port, which tells the control to use the default port number. If the function returns a value other than zero, this indicates an error and a message box is used to display the error to the user.

Property Values

Another method is to initialize the control's properties and then call the **Connect** method without any arguments. Here is an example of how that could be done:

```
Dim nError As Long

HttpClient1.HostName = Text1.Text
HttpClient1.UserName = Text2.Text
HttpClient1.Password = Text3.Text

nError = HttpClient1.Connect()
If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

Note that the **RemotePort** property is not specified, which means that the default port number should be used. As with the previous example, if the connection fails then the method will return an error code and a message box is displayed to the user.

Downloading Resources

A web resource is a general term that applies to any document, image or other file which can be accessed through a web server. It also refers to applets or scripts which can be downloaded and executed on the client, or programs which are executed on the server and return data to the client.

The SocketTools HTTP control provides several methods which makes it easy to access a resource on a web server, and either store that resource in memory or as a file on the local system. For the first example, consider a program which requests some data from a server and stores the response in a string:

```
Dim strDocument As String
Dim nLength As Long
Dim nError As Long

HttpClient1.URL = "http://api.sockettools.com/test"

nError = HttpClient1.Connect()
If nError = 0 Then
    nError = HttpClient1.GetData(HttpClient1.Resource, strDocument, nLength)
    HttpClient1.Disconnect
End If
```

In this example, the **URL** property is assigned to the complete URL of the resource to retrieve and then the **Connect** method is called without any arguments. This tells the control that you want to use the information specified in the URL rather than explicitly specify the host name, port number and so on. If the **Connect** method returns zero, then that means no error has occurred, so the next step is to call the **GetData** method.

The first argument to the **GetData** method is the resource portion of the URL, which is returned by the **Resource** property. The second argument is the string buffer that will contain the data when the method returns. The **GetData** also supports two more arguments. An optional third argument is a variable which will contain the amount of data copied into the string buffer. It is not required that you specify this argument, and is included primarily as a convenience.

The optional fourth argument is a numeric value which determines if text resources should be automatically converted to use the standard Windows conventions for text files. This can be important because the default behavior for the control is to return the resource data exactly as it is sent by the server. For text-based resources like HTML documents, this can present a problem if the document on the server uses a different convention to indicate the end-of-line. On UNIX based servers, the end-of-line character is a single linefeed, while on Windows based systems, the end-of-line is a carriage-return and linefeed pair. Rather than writing code to go through the data and perform that conversion if necessary, you can tell the **GetData** method that you want it to automatically convert any text resources. For example:

```
nError = HttpClient1.GetData(HttpClient1.Resource, strDocument, _
                             nLength, httpTransferConvert)
```

In this case, if the resource is a text document, then it will automatically convert the data to use the Windows conventions for text files if necessary. It is important to note that if the resource is not a text file (for example, an image file) then this option does nothing. That being the case, why not perform the conversion as the default? The reason is that many web servers are configured to treat resources which don't have a known MIME content type as text by default. For example, consider a resource like "/files/record.dat" where there isn't a standard MIME type for a file with a .dat extension. In this case, many web servers will incorrectly tell the client that the resource is text, even if that file actually contains non-text data. If the **GetData** method automatically performed this conversion, then it could potentially corrupt the data. By making the conversion an explicit option, it allows you to tell the control that you know the resource you're requesting is textual and should be converted. Otherwise, it simply provides you with the data exactly as it was returned by the server.

If you want to store the resource on the local system rather than in a buffer in memory, then you can use the **GetFile** method instead of **GetData**. The code would be virtually identical except for the different method call:

```
Dim nError As Long

HttpClient1.URL = "http://sockettools.com/"

nError = HttpClient1.Connect()
If nError = 0 Then
    nError = HttpClient1.GetFile(strFileName, HttpClient1.Resource)
    HttpClient1.Disconnect
End If
```

The **GetFile** method requires two arguments, the name of the file to create or overwrite on the local system and the resource to retrieve from the server. There is also an optional third argument which allows you to specify if text resources should be automatically converted, just as with the **GetData** method.

If you would like to provide some visual feedback to your user with the status of the resource being downloaded, you can implement an event handler for the **OnProgress** event. For example, if you had a standard ProgressBar control on the form, you could simply write code such as this:

```
Private Sub HttpClient1_OnProgress(ByVal BytesTotal As Variant, _
                                   ByVal BytesCopied As Variant, _
                                   ByVal Percent As Variant)
    ProgressBar1.Value = Percent
End Sub
```

As the resource is being retrieved using either the **GetData** or **GetFile** methods, the **OnProgress** event will be periodically fired. In this case, the ProgressBar control is updated using the percentage of the transfer that has completed.

Executing Scripts

A script is simply another resource on the web server which you can access using the HTTP control. Scripts may be complete programs written in a language like Perl, or they may be scripting code intermixed with HTML, such as with ASP and PHP. In most cases, the scripts either generate HTML output which is typically displayed in a browser or XML which will be parsed by the client.

The two most common ways to execute a script is to use either the **GetData** or **PostData** methods. In the examples that we'll be using the script will be written in Perl, however this information applies to ASP and PHP as well. Executing a script that does not require any input is as simple as retrieving an HTML document:

```
Dim strOutput As String
Dim nError As Long

HttpClient1.URL = "http://sockettools.com/cgi-bin/test.cgi"

nError = HttpClient1.Connect()
If nError = 0 Then
    nError = HttpClient1.GetData(HttpClient1.Resource, strOutput)
    HttpClient1.Disconnect
End If
```

In this example, the script /cgi-bin/test.cgi is executed on the server, and the HTML output that it generates is returned to the program in the strOutput variable. The the first few lines of HTML returned by the script looks like this:

```
<html>
<head>
<title>Test Script</title>
<meta http-equiv="pragma" content="no-cache">
</head>
<body>
<h3>Query Parameters</h3>
No query parameters were passed to this script<br>
<h3>Form Variables</h3>
No form variables were passed to this script<br>
<h3>Environment Variables</h3>
<b>DOCUMENT_ROOT</b> = "/var/www/html"<br>
<b>GATEWAY_INTERFACE</b> = "CGI/1.1"<br>
```

Note that near the beginning, there are the lines "No query parameters were passed to this script" and "No form variables were passed to this script". This is because this test script is also capable of displaying any data passed to the script in the form of parameters or form variables. These are the two methods that can be used by a client to provide a script with additional information. The way that the script processes that information, and which method is used, is determined by how the script is written. This is an important point when developing client applications that interact with web scripts. As the client, you need to know what data the script expects and how it expects that data to be passed to it, either as one or more parameters or as form data which would typically be entered using a web browser.

Query Parameters

A script can have data passed to it through one or more parameters which are provided along with the name of the script to be executed. For example, if you wanted to pass two parameters to our test script named "param1" and "param2" with the values of "value1" and "value2" respectively then the URL would look like this:

```
http://sockettools.com/cgi-bin/test.cgi?param1=value1&param2=value2
```

The question mark separates the name of the resource from its arguments. Each argument consists of a name-value pair, separated by an equal sign. If there is more than one name-value pair, then they are separated by

an ampersand. If we modify the example above to use this URL with the query parameters, the first few lines of output returned by the script now looks like this:

```
<html>
<head>
<title>Test Script</title>
<meta http-equiv="pragma" content="no-cache">
</head>
<body>
<h3>Query Parameters</h3>
<b>param1</b> = "value1"<br>
<b>param2</b> = "value2"<br>
<h3>Form Variables</h3>
No form variables were passed to this script<br>
<h3>Environment Variables</h3>
<b>DOCUMENT_ROOT</b> = "/var/www/html"<br>
<b>GATEWAY_INTERFACE</b> = "CGI/1.1"<br>
```

You'll notice that the Query Parameters heading now lists the two arguments with their name-value pairs. Passing query parameters to a script is the easiest method a client can use to provide information to that script. However, most servers have a limit on the maximum length of a URL including any parameters; that value is typically around 8,192 bytes, however it can vary from server to server. Although query parameters are convenient when the script does not require a lot of data, there needs to be a method where larger amounts of data can be provided. This is where form data comes in.

Form Data

When an HTML page presents a form to the user with text boxes, dropdown lists and so forth, that data is typically submitted to a script that is specified by the <form> element. For example, let's consider a simple HTML form:

```
<form action="http://sockettools.com/cgi-bin/test.cgi" method="post">
  <input type="text" name="data1" value=""><br>
  <input type="text" name="data2" value=""><br>
  <input type="submit">
</form>
```

The <form> element provides you two very important pieces of information. The first is the name of the script that will process the data entered by the user. This is specified by the action attribute. The second is how the data will be passed to the script, and that's specified by the method attribute. If the form uses the "get" method, then you'll want to use the control's GetData method as described in the previous section. However, if the form uses the "post" method, then you'll need to use the control's PostData method instead.

In the HTML form, there are two text input fields named "data1" and "data2". For the next example, let's say that you want to write a program that submits data to the script as though the user entered the string "testing1" in the first text box, and "testing2" in the second and then clicked the submit button.

```
Dim strOutput As String
Dim strFormData As String
Dim nError As Long

HttpClient1.URL = "http://sockettools.com/cgi-bin/test.cgi"
strFormData = "data1=testing1&data2=testing2"

nError = HttpClient1.Connect()
If nError = 0 Then
    nError = HttpClient1.PostData(HttpClient1.Resource, strFormData, strOutput)
    HttpClient1.Disconnect
End If
```

You'll notice that the code is substantially similar to the example that uses **GetData**, with the exception that the form data is passed as a separate argument to the **PostData** method. The same convention applies, with the name-value pairs be separated by an equal sign, and multiple pairs being separated by an ampersand. When the **PostData** method is used, the first few lines of output that would be returned would look like this:

```
<html>
<head>
<title>Test Script</title>
<meta http-equiv="pragma" content="no-cache">
</head>
<body>
<h3>Query Parameters</h3>
No query parameters were passed to this script<br>
<h3>Form Variables</h3>
<b>data1</b> = "testing1"<br>
<b>data2</b> = "testing2"<br>
<h3>Environment Variables</h3>
<b>CONTENT_LENGTH</b> = "29"<br>
<b>CONTENT_TYPE</b> = "application/x-www-form-urlencoded"<br>
<b>DOCUMENT_ROOT</b> = "/var/www/html"<br>
<b>GATEWAY_INTERFACE</b> = "CGI/1.1"<br>
```

Now, instead of there being query parameters, the script reports that there are two form variables, "data1" and "data2" and it displays their values. This is the same output that you would get from clicking on the submit button in the HTML form.

Email Services

Electronic mail is the most prevalent application in computer networking and its use has evolved beyond the simple exchange of text messages between two people. For the developer, email provides a reliable means for sending and receiving messages where the protocols are based on well-known and widely used standards. SocketTools provides an interface to email services, allowing developers to easily implement this functionality in their own software without requiring general knowledge of network programming or specific application protocols.

This section of the Quick Start Guide will cover a wide range of topics, from how email messages are structured, to how messages are delivered through relay servers. Even if you are only interested in a specific topic, such as how to send a message, we recommend that you read through this complete section so that you have a full understanding of how the various SocketTools controls are designed to work together.

The SocketTools ActiveX Edition consists of five controls which can be used to build a wide range of applications which use email services. Those controls are:

- Domain Name Service Control
- Internet Message Access Protocol Control
- Mail Message Control
- Post Office Protocol Control
- Simple Mail Transfer Protocol Control

Each of these controls implements a specific aspect of sending and receiving email messages, or managing a user's messages on the mail server. The Domain Name Service (DNS) control is typically used in conjunction with the Simple Mail Transfer Protocol (SMTP) control to send email messages to other users. The Internet Message Access Protocol v4 (IMAP4) and Post Office Protocol v3 (POP3) controls are used to manage the messages that a user has received. The Mail Message (MIME) control is used to compose new messages, parse stored messages and modify them if needed. When combined together, these components form the foundation of any complete email based application.

Protocol Standards

There are several core standards which form the foundation for sending and receiving email messages over the Internet and corporate intranets. These standards are defined in documents called RFCs (Request For Comments) which describe how the various protocols should be implemented. The following standards were used when implementing the email related controls in the SocketTools ActiveX Edition:

RFC 822 documents the basic structure of email messages, including how messages should be formatted and what the standard message header fields are. RFC 2045 documents Multipurpose Internet Mail Extensions (MIME), which details how more complicated messages are structured. File attachments, HTML formatted messages and other more complex aspects of message composition are covered by the MIME standard. The Internet Mail control supports both RFC 822 and MIME formatted email messages, including multipart messages which contain alternate text and file attachments.

RFC 1939 documents the Post Office Protocol v3 (POP3) which is used to retrieve messages from a user's mailbox on a server. The Internet Mail control uses this protocol to enable applications to list, retrieve and delete messages.

RFC 3501 documents the Internet Message Access Protocol v4 (IMAP4) which is used to manage information in a user's mailbox on the server. Unlike the Post Office Protocol, where messages are downloaded and processed on the local system, the messages on an IMAP4 server are retained on the server and processed remotely. This is ideal for users who need access to a centralized store of messages or have limited bandwidth.

RFC 821 documents the Simple Mail Transfer Protocol (SMTP) which is used to deliver messages to one or more recipients. RFC 1869 documents extensions to the protocol which provide additional services such as delivery status notification and authentication. The Internet Mail control implements both the standard and extended SMTP protocols.

Composing Messages

The SocketTools Mail Message control is designed to give the developer access to the internal structure of a mail message and enable you to easily create new messages or modify existing messages. To understand how this control can be used, it's useful to understand how a message is actually formatted. Here is an example of a simple, plain text email message:

```
From: John Doe <johndoe@company.com>
To: Jane Doe <janedoe@company.com>
Date: Mon, 1 Jul 2002 12:00:00 -0800 (PST)
Subject: Meeting scheduled for next week
Message-ID: <20020601200000.15637@mail01.company.com>
MIME-Version: 1.0
Content-Type: text/plain; charset=utf-8

I wanted to confirm that you would be able to attend the meeting.
If there are any scheduling conflicts, please let me know.
```

The first thing that is apparent is that the message has two discrete sections. The first section consists of one or more header fields, followed by a colon and then a value. The second section contains the body of the message, with the headers and body separated by a single blank line.

Therefore, using this example message, reading the control's **From** property would return the string "John Doe <johndoe@company.com>", which is the address of the person who sent the message. To change the From header field, simply set the **From** property to a new string value.

The following is list of the most commonly used properties read, create or modify a message:

Property	Description
Attachment	The name of a file attachment in the current message part.
Bcc	One or more message recipients (blind carbon copy).
Cc	One or more message recipients (carbon copy).
Content-ID	The content identifier for the current message part.
ContentLength	The size of the current message part in bytes.
ContentType	The content type for the current message part.
Date	The date for the current message.
Encoding	The encoding type for the current message part.
From	The sender of the message.
Localize	Enable or disable message localization.
Mailer	The name of the application that generated the message.
Message	The complete message, including headers and body.
MessageID	A unique identifier string from the current message.
Organization	The name of the sender's organization or company.
Part	The current message part in a multipart message.
PartCount	The number of parts in a multipart message.
Priority	The current message priority.
Recipient	The address of one of the message recipients.
Recipients	The number of recipients for the current message.
ReplyTo	The address to which replies should be sent.
Subject	The subject of the current message.
Text	The text in the current message part.
TimeZone	The current timezone offset for the local system.
To	One or more message recipients.

Most of the message-related properties correspond to specific header fields, such as To, From and Subject. Reading those properties return their respective header values while setting them changes their value in the current message.

For more complex message processing such as attaching files or creating multipart messages, there are a number of additional methods which can be used to manage the current message:

Method	Description
AppendMessage	Append text to the current message.
AttachFile	Attach a file to the current message.
ClearMessage	Clear the contents of the current message.
ComposeMessage	Compose a new message.
CreatePart	Create a new message part in a multipart message.
DeleteHeader	Delete a header from the message.
DeletePart	Delete a message part.
ExportMessage	Export the complete message to a text file.
ExtractFile	Extract a file attachment.
GetFirstHeader	Return the first header in the current message part.
GetHeader	Return the value of a specified header field.
GetNextHeader	Return the next header in the current message part.
ImportMessage	Import a message from a text file.
ParseMessage	Parse a string, adding the contents to the current message.
SetHeader	Set the value of the specified header field.

The header related methods such as **GetHeader** and **SetHeader**, enable an application to read, create or modify any header field regardless of whether or not there is a predefined property value for it. Because there can be a potentially unlimited number of header fields in a message, these methods give the developer more control over the header portion of the message.

New messages can be created by setting properties which comprise the message. Here is an example which would create a short message:

```
MailMessage1.From = "johndoe@company.com"
MailMessage1.To = "janedoe@company.com"
MailMessage1.Date = Date
MailMessage1.Subject = "This is the message subject"
MailMessage1.Text = "This is an example of a new message"
```

The resulting message would look like this:

```
From: johndoe@company.com
To: janedoe@company.com
Date: Fri, 01 Nov 2002 12:00:00 -0800
Subject: This is the message subject
MIME-Version: 1.0
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: 8bit

This is an example of a new message.
```

Note that in addition to those properties that were set, there were a number of additional header fields such as MIME-Version and Content-Type that were automatically created.

Although setting properties is one way to create a new message, it involves writing a fair amount of code. There is a simpler way to do it using a single method called **ComposeMessage**. The equivalent code would look like this:

```
MailMessage1.ComposeMessage "johndoe@company.com", _
                            "janedoe@company.com", , , _
                            "This is the message subject", _
                            "This is an example of a new message"
```

The **ComposeMessage** method has the following arguments:

Property	Description
From	A string value which specifies the email address of the person sending the message. This argument is required.
To	A string value which specifies one or more email addresses of those who will receive the message. Multiple addresses may be separated by a comma (such as "johndoe@company.com, janedoe@company.com"). This argument is required.
Cc	An optional string value which specifies recipients who should receive a copy of the message. Multiple recipients may be separated by a comma and the addresses are included in the header of the message.
Bcc	An optional string value which specifies recipients who should receive a copy of the message, however, these addresses are not included in the header of the message. Multiple recipients may be separated by a comma.
Subject	An optional string value which specifies the subject of the message. If the argument is not specified then the message is created without a subject.
MessageText	An optional string value which specifies the body of the message. If the argument is not specified, then the message is created without a body.
MessageHTML	An optional string value which specifies an HTML version of the message. If this argument is provided along with the ***MessageText*** argument, then a multipart message is created which contains both plain text and HTML versions of the message. If the ***MessageText*** argument has not been specified, then only an HTML message is created. If this argument is omitted, then the message is sent with only a plain text body.
CharacterSet	An optional integer value which specifies a character set to use when composing the message. This typically only needs to be set for languages which use extended characters. For more information on the available character sets, consult the technical reference.
EncodingType	An optional integer value which specifies an encoding type to be used with the character set that was selected. For more information about the encoding types available, consult the technical reference.

Once the message has been created, it can be further modified by setting properties or calling methods such as **SetHeader**. Note that you are not restricted to changing only certain header fields. You can create, modify or delete any header in the message that you wish. You can also add your own custom header fields if you wish.

Now that a simple message has been created, let's attach a file to the message. This can be easily done using the **AttachFile** method:

```
MailMessage1.AttachFile "c:\temp\image.gif"
```

Although this is a simple operation, it makes some significant changes to the message (some portions of the attachment data has been omitted):

```
From: johndoe@company.com
To: janedoe@company.com
Date: Fri, 01 Nov 2002 12:00:00 -0800
Subject: This is the message subject
MIME-Version: 1.0
Content-Type: multipart/mixed;
              boundary="------=_ST4020_0001_0BCF2D17_179E5A2E"
Content-Transfer-Encoding: 8bit

This is a multipart message in MIME format.

------=_ST4020_0001_0BCF2D17_179E5A2E
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: 8bit

This is an example of a new message.
------=_ST4020_0001_0BCF2D17_179E5A2E
Content-Type: image/gif
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="image.gif"
Content-Length: 6434

R01GOD1hRgEyAPcAAJaWqqKitp6esqamuqamtrKywq6uvsrK176+yrq6xra2wsbG0sLCztLS
287O18LCytvb49fX38rK0sbG2t/f5/f3+/Pz9+/v8+vr7+fn6+Pj59/f49vb311h1hhmFl
uXvrtV8rfWQ712qbuZ3rt3rLFomFovruR0hq4bbuZwbu1hLubBbuZF7tn7WuE2RurUwu7DL
u3nrt7KbrJLbMvPmu6hvFC2atcfburarucL7uY/Lur1TuVpLu5CLvJyLu9obvZO7u900W7jA
K7jkW72maxHGC72kWxbAu7xuq77DS71p67qg177va73FqxGryxDMK7/m67zgixEBAQA7

------=_ST4020_0001_0BCF2D17_179E5A2E--
```

The message has now become a multipart message that contains both human-readable text as well as data for the file attachment. Rather than having a single group of headers followed by a message body, the message is now broken into sections, each with its own group of headers and body. Each of these sections are called a message part, and can be accessed individually using the **Part** property. Each message part is identified by a part number which starts at zero and increases for each subsequent part. Part 0 of this message consists of the following:

```
From: johndoe@company.com
To: janedoe@company.com
Date: Fri, 01 Nov 2002 12:00:00 -0800
Subject: This is the message subject
MIME-Version: 1.0
Content-Type: multipart/mixed;
              boundary="------=_ST4020_0001_0BCF2D17_179E5A2E"
Content-Transfer-Encoding: 8bit

This is a multipart message in MIME format.
```

Part 0 of any message always refers to the headers and body of the main message. In the previous message, part 0 contains the entire message. Here, part 0 consists primarily of headers and a brief message that this is now a multipart message. This is automatically done for the benefit of older mail clients which may not understand a MIME formatted message, so the user has a message that at least identifies what the message is. Another thing that has changed is the value of the Content-Type header. In the previous message it had a value of "text/plain; charset=utf-8" which tells the mail client that this is a plain text message. With the file attachment, this has changed to a type called "multipart/mixed" which indicates that the message contains multiple parts with mixed types of information. The boundary value is what is used to actually designate the different parts of the message. As the message is being processed, the mail client knows that it has found a new message part when the boundary string is encountered.

The next part of the message, part 1, contains the message that was in the original version of the message:

```
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: 8bit
This is an example of a new message
```

Note that the content type is back to plain text, just as it was with the original. When a mail client processes a message, it scans the message for plain text message parts which contain information to be displayed to the user.

The last part of the message, part 2, contains the actual file data that was attached to the message:

```
Content-Type: image/gif
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="image.gif"
Content-Length: 6434

R01GOD1hRgEyAPcAAJaWqqKitp6esqamuqamtrKywq6uvsrK176+yrq6xra2wsbG0sLCztLS
287O18LCytvb49fX38rK0sbG2t/f5/f3+/Pz9+/v8+vr7+fn6+Pj59/f49vb311h1hhmFl
uXvrtV8rfWQ712qbuZ3rt3rLFomFovruR0hq4bbuZwbu1hLubBbuZF7tn7WuE2RurUwu7DL
u3nrt7KbrJLbMvPmu6hvFC2atcfburarucL7uY/Lur1TuVpLu5CLvJyLu9obvZO7u900W7jA
K7jkW72maxHGC72kWxbAu7xuq77DS71p67qg177va73FqxGryxDMK7/m67zgixEBAQA7
```

Here the Content-Type header tells the mail client that this is an image file in the GIF format. The other header fields in this message part are used by applications to extract the file attachment once it has been delivered to the recipient. Because email messages must be sent over systems which may not be able to handle binary data, the image file data has been ***encoded*** using a standard algorithm called base64. This algorithm converts binary data into plain 7-bit text data that can be safely exchanged with other mail servers. The process of encoding and decoding attachments is automatically handled by the control when the file is attached. The **ExtractFile** method is essentially the reverse of the **AttachFile** method, automatically decoding and storing a file attachment on the local system.

Composing Text Messages

To compose a simple text message, you can use the **ComposeMessage** method and specify the body of the message, along with the other standard header fields. For example, if you had a form with TextBox controls for the sender, recipients, subject and message body you could use code like this:

```
Dim nError As Long

nError = MailMessage1.ComposeMessage(editFrom.Text, _
                                     editTo.Text, _
                                     editCc.Text, _
                                     editBcc.Text, _
                                     editSubject.Text, _
                                     editMessage.Text)
If nError Then
   MsgBox "Unable to compose a new message" & vbCrLf & _
          MailMessage1.LastErrorString, vbExclamation
   Exit Sub
End If
```

If you have a text file that contains the body of the message that you want to use, then you can create a message without a message body and then read the contents of the file and assign it to the **Text** property. For example:

```
Dim nError As Long

nError = MailMessage1.ComposeMessage(editFrom.Text, _
                                     editTo.Text, _
                                     editCc.Text, _
                                     editBcc.Text, _
                                     editSubject.Text)
If nError Then
   MsgBox "Unable to compose a new message" & vbCrLf & _
          MailMessage1.LastErrorString, vbExclamation
   Exit Sub
End If

' Open the file for and assign the contents of the file to the
' Text property which will put it in the body of the message
hFile = FreeFile()
Open strFileName For Input As hFile
MailMessage1.Text = Input(LOF(hFile), hFile)
Close hFile
```

To access the complete message, use the **Message** property, which will return the complete message including the headers and message body. This is most commonly used with the SMTP control to submit the message to a mail server for delivery to the recipients.

Composing HTML Messages

To compose an HTML formatted message, the **ComposeMessage** method can be used in the same way that text messages are created. For example, consider the following HTML text:

```
<html>
<head></head>
<body>
<font face="Arial">
<h3>Test HTML Message</h3>
This is a test message which uses HTML to format the text. This
message was created using the <b>Mail Message</b> control from
<a href="http://sockettools.com/">Catalyst Development</a>.
</font>
</body>
</html>
```

You could either assign this text to a string, or you could read the message from a file using code like this:

```
hFile = FreeFile()
Open strFileName For Input As hFile
strMessageHTML = Input(LOF(hFile), hFile)
Close hFile
```

Where *strMessageFile* contains the HTML message you wish to send. To compose the HTML formatted email, simply call the **ComposeMessage** method as you would with a plain text message, except that instead of passing the message to the *MessageText* argument, you pass it to the *MessageHTML* argument:

```
nError = MailMessage1.ComposeMessage(editFrom.Text, _
                                     editTo.Text, _
                                     editCc.Text, _
                                     editBcc.Text, _
                                     editSubject.Text, _
                                     "", _
                                     strMessageHTML)
```

The message that will be sent will now be displayed to the recipient using HTML and will include the formatting (such as font and text size) as well as the hyperlink. However, not all mail clients are capable of displaying HTML email. This poses a problem because the message that they'll receive will be the largely unreadable HTML source. To resolve this problem, create both a plain text version of the message along with the HTML version. Ideally it would contain similar content, although you could provide a simple message which says that this is an HTML email and they should request a plain-text version if they can't display HTML messages. In either case, simply provide both the *MessageText* and *MessageHTML* arguments:

```
nError = MailMessage1.ComposeMessage(editFrom.Text, _
                                     editTo.Text, _
                                     editCc.Text, _
                                     editBcc.Text, _
                                     editSubject.Text, _
                                     strMessageText, _
                                     strMessageHTML)
```

This will create what is called a multipart/alternative MIME message which contains both plain text and HTML versions of the message. Mail clients which are capable of displaying the HTML message will use that version, while those that cannot will display the plain text version.

It should be noted that there are still some mail clients which do not understand multipart/alternative messages

and therefore will display both the plain text and the HTML source text. While confusing, the plain text version will ensure that the message is still readable. For the most part, email is still a plain text medium so if you consider readability and compatibility with older mail software to be more important than formatted text, it is recommended that you use only plain text messages. However, if you know that the recipients have mail clients that are capable of displaying HTML, the Mail Message control makes this easy to do.

Importing Messages

In addition to using the **ComposeMessage** method to create a new message, it is possible to import existing messages into the Mail Message control. These messages may exist either as text files already stored on the local system, records in a database, or may even be created dynamically by the application using data from other sources. The most important thing to keep in mind is that any message which is imported into the control must adhere to the basic structure outlined previously in the section discussing message composition. The control is tolerant of malformed messages, however, importing a corrupted message will often produce unexpected results. If the source of the message is unknown, it is recommended that an application perform checks to ensure that it contains reasonable values. For example, check to make sure the From and To header fields contain email addresses, the message has a valid Date, and a message body is present.

The simplest method of importing a message into the control is using the **ImportMessage** method. Here is an example which uses the Common Dialog control to select a file and then calls **ImportMessage** to import the file:

```
Dim nError As Long

On Error GoTo ImportCanceled
CommonDialog1.CancelError = True
CommonDialog1.DefaultExt = ".txt"
CommonDialog1.DialogTitle = "Import Message"
CommonDialog1.FilterIndex = 1
CommonDialog1.Flags = cdlOFNFileMustExist + cdlOFNLongNames
CommonDialog1.Filter = "Text Files (*.txt)|*.txt|" & _
                       "email Message Files (*.eml)|*.eml|" & _
                       "All Files (*.*)|*.*"

CommonDialog1.ShowOpen
On Error GoTo 0

nError = MailMessage1.ImportMessage(CommonDialog1.FileName)
If nError Then
    MsgBox "Unable to import message from " & _
           CommonDialog1.FileTitle & vbCrLf & _
           MailMessage1.LastErrorString, vbExclamation
    Exit Sub
End If

MsgBox "Imported message from " & MailMessage1.From & vbCrLf & _
       "regarding " & Chr(34) & MailMessage1.Subject & Chr(34), _
       vbInformation

Exit Sub

ImportCanceled:
Exit Sub
```

Once the message has been imported successfully, the various message related properties can be accessed just as if the message had been composed. Note that the current message, if any, will be completely replaced by the message that has been imported.

Another method of importing a message is from a string. This is useful if a message has been stored in something other than a text file such as a record in a database. To do this, simply set the control's **Message** property to the string which contains the message:

```
On Error Resume Next: Err.Clear
MailMessage1.Message = strMessage
```

```
If Err.Number Then
    MsgBox Err.Description, vbExclamation
    Exit Sub
End If
```

Unlike the **ImportMessage** method, which returns an error code if it fails, setting the **Message** property will result in an error being raised if there is a problem. Because of this, any application which sets the **Message** property should use an error handler. In this example, it simply executes the next statement and uses the *Err* object to obtain the error code and description.

A third method of importing a message into the control is to use the **ParseMessage** method. Unlike the **ImportMessage** method or the **Message** property, it is not required that the complete message be available at once. Instead, **ParseMessage** enables an application to import a message in pieces, dynamically parsing the data and adding to the contents of the current message. The following example opens a file which contains a message, reads it in 1,024 byte blocks and then passes it to the **ParseMessage** method:

```
hFile = FreeFile()
Open strFileName For Input As hFile
nFileLength = LOF(hFile)

MailMessage1.ClearMessage

Do While nFileLength > 0
    cbBuffer = nFileLength
    If cbBuffer > 1024 Then cbBuffer = 1024
    nFileLength = nFileLength - cbBuffer
    strBuffer = Input(cbBuffer, hFile)
    nError = MailMessage1.ParseMessage(strBuffer)
    If nError > 0 Then
        MsgBox MailMessage1.LastErrorString, vbExclamation
        Exit Do
    End If
Loop

Close hFile
```

The initial call to the **ClearMessage** method ensures that if there is a current message, the contents are cleared first. Then the **ParseMessage** method is repeatedly called until the end-of-file is reached. This approach could be used when a message is being created by some external data source or the application needs to make some sort of change to the message contents dynamically. Note that the purpose of the above example is to demonstrate how to use the **ParseMessage** method and is not the recommended procedure for importing a message from a text file. Refer to the technical reference documentation for the **ImportMessage** method for more information on importing messages from text files.

Exporting Messages

The Mail Message control has the ability to export the current message contents as a string or as a text file on the local system. When a message is exported, the complete message including headers, message body and any file attachments are included. The following example uses the CommonDialog control to choose a file name to export the current message to:

```
Dim nError As Long

On Error GoTo ExportCanceled
CommonDialog1.CancelError = True
CommonDialog1.DefaultExt = ".txt"
CommonDialog1.DialogTitle = "Export Message"
CommonDialog1.FilterIndex = 1
CommonDialog1.Flags = cdlOFNLongNames + cdlOFNOverwritePrompt
CommonDialog1.Filter = "Text Files (*.txt)|*.txt|" & _
                       "email Message Files (*.eml)|*.eml|" & _
                       "All Files (*.*)|*.*"

CommonDialog1.ShowSave
On Error GoTo 0

nError = MailMessage1.ExportMessage(CommonDialog1.FileName)
If nError Then
    MsgBox "Unable to export message to " & _
           CommonDialog1.FileTitle & vbCrLf & _
           MailMessage1.LastErrorString, vbExclamation
    Exit Sub
End If

MsgBox "Exported message to " & CommonDialog1.FileName & vbCrLf & _
       "regarding " & Chr(34) & MailMessage1.Subject & Chr(34), _
       vbInformation

ExportCanceled:
Exit Sub
```

When a message is exported, headers may be re-ordered and certain headers which contain routing information (such as Received and Return-Path) are omitted by default. These headers are not normally needed when composing or delivering a message, however, there may be situations in which an application needs to preserve these headers or the order in which they were originally received. The **ExportMessage** method has an optional argument which can be used to specify one or more export options:

Value	Description
mimeOptionDefault	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
mimeOptionAllHeaders	All headers, including the Bcc, Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.

mimeOptionKeepOrder	The original order in which the message header fields were set or imported are preserved when the message is exported.

The mimeOptionAllHeaders and mimeOptionKeepOrder values may be combined if both options are required. For example, the following code would export a message with all of the headers, preserving their original order:

```
nOptions = mimeOptionAllHeaders Or mimeOptionKeepOrder
nError = MailMessage1.ExportMessage(strFileName, nOptions)
```

Note that the option values are actually bit flags, so a bitwise Or operation is used to combine them. This method is preferred over using simple addition which can produce unexpected results in some cases. Also note that if the **ExportMessage** method is called without specifying the optional argument, then the value of the **Options** property is used as a default, such as:

```
MailMessage1.Options = mimeOptionAllHeaders Or mimeOptionKeepOrder
nError = MailMessage1.ExportMessage(strFileName)
```

This would also cause the message to be exported with all headers and preserve their original order. A general rule of thumb is that if there is an optional argument to a method which corresponds to a property in the control, if that argument is not specified, the property value will be used as a default.

If an application needs to do some processing on the message but doesn't want the overhead of exporting the message to a file, then the message contents can be read using the control's **Message** property. This property returns a string which contains the complete message, including all headers. The **Options** property determines whether or not all headers are exported and if the original header order is preserved, just as with the **ExportMessage** method. It should be noted that this is different than the **Text** property, which returns only the body of the current message part, not the complete message.

File Attachments

In addition to sending text messages, email is commonly used as a means to exchange files. This can be easily done using the Mail Message control's **AttachFile** method. Let's modify the previous example, presuming that an edit control has been included on the form which allows a user to input the name of the file they wish to attach. Add this after the call to the **ComposeMessage** method:

```
' If a file name has been entered, then attach it to
' the message that was composed

If Len(editFileName.Text) > 0 Then
    nError = MailMessage1.AttachFile(editFileName.Text)

    If nError Then
        MsgBox "Unable to attach file " & editFileName.Text & _
               vbCrLf & MailMessage1.LastErrorString, vbExclamation
    Exit Sub
End If
```

If the file does not exist or cannot be accessed, then the **AttachFile** method will return an error. Otherwise, the file data will be encoded and attached to the message. The **AttachFile** method has two arguments, the name of the file to attach and an optional argument which specifies how the attachment should be encoded.

In most cases, it is not necessary to specify the optional argument because the **AttachFile** method will automatically determine the correct encoding method based on the contents of the file. However, there are some situations in which you may wish to use some specific encoding method. For example, you may want to force the control to use base64 encoding even though the attachment is a plain text file. To do this, you can use one of the following values:

Value	Description
mimeAttachBase64	The base64 algorithm is used to encode the file data. This is the default encoding type used for binary data such as executables, image or audio files.
mimeAttachUucode	The uuencode algorithm is used to encode the file data. This is an older encoding type that was commonly used before the MIME standard was developed. It is not recommended that you use this encoding method unless specifically required by an application.
mimeAttachQuoted	The quoted-printable algorithm is used to encode the file data. This should only be used to encode text files which may contain non-printable or extended ANSI characters. Using this format on binary files may cause them to become corrupted when extracted by the recipient.

For example, if you want to always have the attached file encoded using the base64 algorithm, the code would be changed to look like this:

```
' If a file name has been entered, then attach it to
' the message that was composed

If Len(editFileName.Text) > 0 Then
    nError = MailMessage1.AttachFile(editFileName.Text, mimeAttachBase64)
    If nError Then
        MsgBox "Unable to attach file " & editFileName.Text & _
               vbCrLf & MailMessage1.LastErrorString, vbExclamation
    Exit Sub
End If
```

When attaching a file, keep in mind that the size of the attachment in the message will typically be about 33% larger than the size of the file itself. This is an important consideration because most mail servers restrict the size of the messages they will accept and will reject messages that exceed that limit. For example, if a mail server restricts messages to 5 megabytes, the maximum size of a file that can be attached to the message is about 3.5 megabytes.

Another consideration with file attachments is compatibility with third-party mail client software. If the current message contains alternative messages (i.e., both plain text and HTML text) then **AttachFile** will change the message structure, creating a more complex multipart message which has mixed content types. Mail software which does not fully conform to the MIME standard may not be able to correctly display this type of message, either being unable to display the body of the message, or, displaying the complete message including the alternate text and the encoded file attachment. To ensure that your message is readable by most recipients, it's recommended that you attach files to plain text messages.

Sending Messages

After a mail message has been created or imported, that message can be delivered to the recipients using the Simple Mail Transfer Protocol (SMTP) control. To send a message, simply follow these steps:

- Create or import a message to be submitted for delivery
- Create a list of one or more recipients for the message
- Connect to the mail server, authenticating if needed
- Submit the message to the mail server for delivery
- Disconnect from the mail server

In the following example, the Mail Message control is used to compose a message and the SMTP control is used to submit the message to a mail server for delivery. Typically the name of the mail server would be provided by the user and would be the server provided by their Internet Service Provider.

```
With MailMessage1
    ' Create a standard mail message with the sender, recipients,
    ' subject and message body
    .ComposeMessage strFrom, strTo, strCc, , strSubject, strMessage

    ' Create a string which contains a comma separated list of all of
    ' the message recipients for use with the SendMessage method
    For nIndex = 0 To .Recipients - 1
        If Len(strRecipients) > 0 Then strRecipients = strRecipients & ", "
        strRecipients = strRecipients & .Recipient(nIndex)
    Next

    ' Connect to the mail server and display a message box if the
    ' connection fails for any reason
    nError = SmtpClient1.Connect(strServer)
    If nError Then
        MsgBox SmtpClient1.LastErrorString, vbExclamation
        Exit Sub
    End If

    ' Submit the message to the mail server
    nError = SmtpClient1.SendMessage(.From, strRecipients, .Message)
    If nError Then
        MsgBox SmtpClient1.LastErrorString, vbExclamation
        SmtpClient1.Disconnect
        Exit Sub
    End If

    ' Disconnect from the mail server after the message has
    ' been submitted for delivery
    SmtpClient1.Disconnect
End With
```

Client Authentication

In some cases a mail server may require that the client authenticate itself before it is permitted to submit a message for delivery. This is typically done as an anti-spam measure to ensure that only authorized users are able to send messages. Authentication is part of the Extended SMTP (ESMTP) standard, so to use this feature it is required that you set the **Extended** property to True, and then call the **Authenticate** method after the connection has been established. For example:

```
SmtpClient1.Extended = True

nError = SmtpClient1.Connect(strHostName)
If nError Then
    MsgBox SmtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

If SmtpClient1.Extended = False Then
    MsgBox "This server does not support authentication", vbInformation
    SmtpClient1.Disconnect
    Exit Sub
End If

nError = SmtpClient1.Authenticate(strUserName, strPassword)
If nError Then
    MsgBox SmtpClient1.LastErrorString, vbExclamation
    SmtpClient1.Disconnect
    Exit Sub
End If
```

First, the **Extended** property is set to True, which tells the SMTP control that we would like to try and establish an ESMTP session. After the connection has been established, the value of the **Extended** property is checked to make sure that it is still True. Because ESMTP is an optional extension to the protocol, it is not required that all mail servers support it. The control will not return an error if the server doesn't support ESMTP, it will simply set the **Extended** property back to False, which lets the application know that the connection has been made, but no extended features are available.

The **Authenticate** method expects two arguments, a username and password which are used to authenticate the client session. Once the session has been authenticated, the client can proceed to submit the message for delivery.

Relay Servers

In some situations it may not be possible to send mail directly to the server that accepts mail for a given domain. The two most common situations are corporate networks which have centralized servers that are responsible for delivering and forwarding messages, or an Internet Service Provider (ISP) which specifically blocks access to all mail servers other than their own. This is usually done as either a security measure or as a means to inhibit users from sending unsolicited commercial email messages. If the standard SMTP port is being blocked, then any connection attempts will either fail immediately with an error that the server is unreachable, or the connection will simply time-out. In either case, a relay server must be specified in order to send email messages.

A relay server is a system which will accept messages addressed to users which may be in a different domain, and will relay those messages to the appropriate server that does accept mail for the domain. In that case, the **HostName** property will specify the host name or address of the mail server that will be used by the control to relay the message. In some cases, the relay server may use a non-standard port number to circumvent blocks on port 25. In that case, the **RemotePort** property should be set to the port number used by the relay server.

It is important to note that using a mail server as a relay without the permission of the organization or individual who owns that server may violate Acceptable Use Policies and/or Terms of Service agreements with your service provider. Systems which relay messages from anyone, regardless of whether the message is coming from a recognized domain, are called ***open relays***. Because open relays are often used to send unsolicited email, many administrators block mail that comes from one. It is recommended that users check with their network administrators or Internet service providers to determine if access to external mail servers is restricted and what is the acceptable use policy for relaying messages through their mail servers.

Listing Messages

If your application needs to list the available messages in a user's mailbox, either the Post Office Protocol v3 (POP3) or Internet Message Access Protocol v4 (IMAP4) controls can be used. Which protocol is selected largely depends on the mail server. Most service providers offer POP3 access to their mail servers, and it is the more commonly used protocol.

Post Office Protocol

The Post Office Protocol is a simple message retrieval protocol designed to access the messages that have arrived in the user's inbox. It is designed for mail clients which will store the messages on the local system and then delete them from the server. POP3 provides only limited support for managing messages on the server. Here is an example of how you could list messages using the POP3 and Mail Message controls:

```
Dim nMessageId As Long
Dim strHeaders As String
Dim nError As Long

' Establish a connection to the mail server using the default port
' and the specified username and password
nError = PopClient1.Connect(strHostName, , strUserName, strPassword)
If nError Then
    MsgBox PopClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' If the MessageCount property returns 0, then there are no
' messages in the mailbox
If PopClient1.MessageCount = 0 Then
    MsgBox "There are no messages in this mailbox", vbInformation
    PopClient1.Disconnect
    Exit Sub
End If

' Initialize the ListBox and ProgressBar controls
List1.Clear
ProgressBar1.Max = PopClient1.MessageCount - 1
ProgressBar1.Value = 0

For nMessageId = 1 To PopClient1.MessageCount
    ' Retrieve the headers for the message, storing them in the
    ' string variable
    nError = PopClient1.GetHeaders(nMessageId, strHeaders)
    If nError Then Exit For

    ' Parse the header block returned by the server
    MailMessage1.ClearMessage
    MailMessage1.ParseMessage strHeaders

    ' Update the ListBox and ProgressBar controls
    List1.AddItem nMessageId & " " & MailMessage1.Subject
    ProgressBar1.Value = nMessageId - 1
    DoEvents
Next

If nError Then MsgBox PopClient1.LastErrorString, vbExclamation
PopClient1.Disconnect
```

There's a few important points about this example. The use of the **GetHeaders** method here to retrieve the

message header block is the most compatible means of retrieving header values from a POP3 server. If you review the Technical Reference, you'll notice that there is also a **GetHeader** method which will return the value of a single header field. While it may seem more efficient to use this approach, rather than download the complete header block, the **GetHeader** method depends on an extended command called XTND XLST which many POP3 servers do not support. While it is convenient if the server does support that command, an application should never depend on it being available. In addition, although this example only lists the subject of the message, if you wanted to list more information such as the sender, recipient and date of the message then you'd need to call **GetHeader** multiple times. It is usually more efficient to simply request the entire header block and let the client parse it.

The header block is returned in a string with each header terminated by a carriage return and linefeed sequence. While you could write your own code to parse the headers, it's recommended that you use the Mail Message control to do this. Message headers can be complex when they span multiple lines or contain encoded sequences.

One other thing that is worth pointing out is the use of **DoEvents** in the For..Next loop. This is done to allow the UI controls to redraw themselves and also permit the application to remain responsive to the user. However, when you do this you need to be aware that your application can be re-entered by the user clicking on buttons, selecting menu items and so on. If you use **DoEvents**, make sure that you design your application so that the user can not interact with your program in a way that will allow them to perform some other action using the POP3 control while the contents are being listed.

Internet Message Access Protocol

The Internet Message Access Protocol is a more complex, general purpose protocol used for accessing and managing a user's mailbox on a server. The design of the IMAP4 control is intentionally very similar to the POP3 control, with a number of additional properties and methods to take advantage of the protocol's features. Here is how the code would be written to list the available messages in a user's Inbox:

```
Dim nMessageId As Long
Dim strSubject As String
Dim nError As Long

' Establish a connection to the mail server using the default port
' and the specified username and password
nError = ImapClient1.Connect(strHostName, , strUserName, strPassword)
If nError Then
    MsgBox ImapClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' Select the user's Inbox where new messages arrive
nError = ImapClient1.SelectMailbox("Inbox")
If nError Then
    MsgBox ImapClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' If the MessageCount property returns 0, then there are no
' messages in the mailbox
If ImapClient1.MessageCount = 0 Then
    MsgBox "There are no messages in this mailbox"
    ImapClient1.Disconnect
    Exit Sub
End If

' Initialize the ListBox and ProgressBar controls
```

```
List1.Clear
ProgressBar1.Max = ImapClient1.MessageCount - 1
ProgressBar1.Value = 0

For nMessageId = 1 To ImapClient1.MessageCount
    ' Retrieve the Subject header field from the message
    strSubject = ""
    ImapClient1.GetHeader nMessageId, 0, "Subject", strSubject

    ' Update the ListBox and ProgressBar controls
    List1.AddItem nMessageId & " " & strSubject
    ProgressBar1.Value = nMessageId - 1
    DoEvents
Next

If nError Then MsgBox ImapClient1.LastErrorString, vbExclamation
ImapClient1.Disconnect
```

You'll notice that the code is substantially similar to the previous example, however there are a couple of important differences. The **SelectMailbox** method is used to explicitly select the user's Inbox, where new messages are stored until they are moved or deleted by the user. Unlike POP3 which only deals with new messages, IMAP4 is designed to manage multiple mailboxes, so you need to specify which mailbox you want to use. The mailbox "Inbox" is a special mailbox defined by the IMAP4 standard where all new messages are stored. You can list the available mailboxes by reading the **Mailbox** property array.

The other significant difference is the use the **GetHeader** method here. This example could have used the **GetHeaders** methods to retrieve the complete header block as in the previous example, however because IMAP4 has support for retrieving specific header values, it was a good way to explain the difference between the protocols. Unlike POP3 which depends on an extension to the protocol to retrieve a single header value, all IMAP4 servers support this capability so it is something that you can take advantage of without concerns about compatibility.

It should be noted that if you plan on retrieving more than two or three header fields from the message, it will usually be more efficient to retrieve the entire header block with the GetHeaders method and then use the Mail Message control to parse it.

Reading Messages

The Post Office Protocol (POP3) and Internet Message Access Protocol (IMAP4) controls can be used to retrieve messages from the mail server for the client to process or store locally. Which protocol is used largely depends on what service the mail server supports. It is also important to remember that POP3 is a protocol that is primarily used with applications that store the messages locally; after a message has been downloaded, it is typically deleted from the server. On the other hand, IMAP4 is a protocol designed for clients that want to leave the messages on the server and manage those messages remotely.

Post Office Protocol

The Post Office Protocol gives you access to all of the messages that have arrived in a user's inbox. To read those messages, they must be downloaded one at a time and you must provide a message number which identifies the message that you wish to retrieve or store. The first message in the mailbox is number one, and it increases in order for each additional message. It is important to note that as new messages arrive and old messages are deleted from the mailbox, message numbers will not refer to the same message from session to session. In other words, if you connect to the server and retrieve message 12, disconnect and then re-connect, it is possible that message 12 will no longer be the same message that you previously downloaded. Your application should never depend on a message number referring to a specific message.

Here is an example of how to retrieve the first message in the user's mailbox:

```
Dim nMessageId As Long
Dim strMessage As String
Dim nError As Long

' Establish a connection to the mail server using the default port
' and the specified username and password
nError = PopClient1.Connect(strHostName, , strUserName, strPassword)
If nError Then
    MsgBox PopClient1.LastErrorString, vbExclamation
    Exit Sub
End If

If PopClient1.MessageCount = 0 Then
    MsgBox "There are no messages in this mailbox"
Else
    nMessageId = 1 ' Retrieve the first message from the mailbox
    nError = PopClient1.GetMessage(nMessageId, strMessage)

    ' If there were no errors, then use the Mail Message control
    ' to parse the message and update the interface
    If nError = 0 Then
        MailMessage1.Message = strMessage
        textFrom.Text = MailMessage1.From
        textTo.Text = MailMessage1.To
        textCc.Text = MailMessage1.Cc
        textDate.Text = MailMessage1.Date
        textSubject.Text = MailMessage1.Subject
        textMessage.Text = MailMessage1.Text
    Else
        MsgBox PopClient1.LastErrorString, vbExclamation
    End If
End If

PopClient1.Disconnect
```

In this example, the **GetMessage** method is used to retrieve the contents of the first message, message

number 1, and store it in a string variable. If successful, then the Mail Message control is used to parse the message. This is easily done by assigning the **Message** property to the string which contains the message.

If you prefer to store the message on the local system, then you can use the **StoreMessage** method instead. It works in exactly the same way, except that instead of specifying a string or byte array, you specify the name of a file which will contain the message. If the file already exists it will be overwritten, otherwise it will be created.

For large messages, it can be useful to provide some sort of feedback to the user to let them know how much of the message has been downloaded. This can be easily done by adding an **OnProgress** event handler to update a ProgressBar control. For example:

```
Private Sub PopClient1_OnProgress(ByVal MessageNumber As Variant, _
                                  ByVal MessageSize As Variant, _
                                  ByVal MessageCopied As Variant, _
                                  ByVal Percent As Variant)
    ProgressBar1.Value = Percent
End Sub
```

As the message is being retrieved from the server, the **OnProgress** event will periodically fire which which cause the ProgressBar control to be updated with a new value.

Internet Message Access Protocol

The Internet Message Access Protocol enables you to access the messages in any of the user's mailboxes. To read those messages, you must specify the message number. The message numbers start with one and increase for each message in the mailbox. As with the Post Office Protocol, your application should never depend on a message number referring to a specific message. Messages that are deleted or moved from one mailbox to another may change the ordering of the messages. To identify messages over multiple client sessions, refer to the **MessageUID** property.

The following example retrieves the first message from the user's Inbox:

```
Dim nMessageId As Long
Dim strMessage As String
Dim nError As Long

' Establish a connection to the mail server using the default port
' and the specified username and password
nError = ImapClient1.Connect(strHostName, , strUserName, strPassword)
If nError Then
    MsgBox ImapClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' Select the user's Inbox where new messages arrive
nError = ImapClient1.SelectMailbox("Inbox")
If nError Then
    MsgBox ImapClient1.LastErrorString, vbExclamation
    Exit Sub
End If

If ImapClient1.MessageCount = 0 Then
    MsgBox "There are no messages in this mailbox"
Else
    nMessageId = 1 ' Retrieve the first message from the mailbox
    nError = ImapClient1.GetMessage(nMessageId, 0, strMessage)

    ' If there were no errors, then use the Mail Message control
    ' to parse the message and update the interface
    If nError = 0 Then
```

```
        MailMessage1.Message = strMessage
        textFrom.Text = MailMessage1.From
        textTo.Text = MailMessage1.To
        textCc.Text = MailMessage1.Cc
        textDate.Text = MailMessage1.Date
        textSubject.Text = MailMessage1.Subject
        textMessage.Text = MailMessage1.Text
    Else
        MsgBox ImapClient1.LastErrorString, vbExclamation
    End If
End If

ImapClient1.Disconnect
```

This example is very similar to the code used with the POP3 control. The only significant changes are the use of the **SelectMailbox** method to select the user's Inbox, and the extra argument to the **GetMessage** method. Unlike the Post Office Protocol, IMAP4 has the ability to select specific sections of a multipart message and return them to the caller.

When working with multipart messages in the IMAP4 control, one important thing to keep in mind is that the IMAP4 protocol considers the first part of a multipart message to be part 1. Referencing part 0 tells the control that you want the entire multipart message including the main header block.

For large messages, it can be useful to provide some sort of feedback to the user to let them know how much of the message has been downloaded. This can be easily done by adding an **OnProgress** event handler to update a ProgressBar control. This is implemented in exactly the same way as it is for the POP3 control. For example:

```
Private Sub ImapClient1_OnProgress(ByVal MessageNumber As Variant, _
                                   ByVal MessageSize As Variant, _
                                   ByVal MessageCopied As Variant, _
                                   ByVal Percent As Variant)
    ProgressBar1.Value = Percent
End Sub
```

As the message is being retrieved from the server, the **OnProgress** event will periodically fire which which cause the ProgressBar control to be updated with a new value.

Deleting Messages

If your application needs to delete the messages in a user's mailbox, either the Post Office Protocol v3 (POP3) or Internet Message Access Protocol v4 (IMAP4) controls can be used. Which protocol is selected largely depends on the mail server. Most service providers offer POP3 access to their mail servers, and it is the more commonly used protocol.

Post Office Protocol

The Post Office Protocol is designed for mail clients which will store the messages on the local system and then delete them from the server. Here is an example of how you could store all of the messages in a user's mailbox and then delete them:

```
Dim nMessageId As Long
Dim strHeaders As String
Dim nError As Long

' Establish a connection to the mail server using the default port
' and the specified username and password
nError = PopClient1.Connect(strHostName, , strUserName, strPassword)
If nError > 0 Then
    MsgBox PopClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' If the MessageCount property returns 0, then there are no
' messages in the mailbox
If PopClient1.MessageCount = 0 Then
    MsgBox "There are no messages in this mailbox", vbInformation
    PopClient1.Disconnect
    Exit Sub
End If

' Initialize the ProgressBar control
ProgressBar1.Value = 0
ProgressBar1.Max = PopClient1.LastMessage

For nMessageId = 1 To PopClient1.LastMessage
    ' Create a file name based on the message number
    strFileName = strFolder + Format(nMessageId, "00000000") + ".eml"

    ' Store the message in the specified file
    nError = PopClient1.StoreMessage(nMessageId, strFileName)
    If nError = 0 Then
        ' If the message was stored successfully, then delete
        ' it from the mailbox
        nError = PopClient1.DeleteMessage(nMessageId)
    End If

    ' If there was an error, warn the user
    If nError > 0 Then
        MsgBox PopClient1.LastErrorString, vbExclamation
        Exit For
    End If

    ProgressBar1.Value = nMessageId
    DoEvents
Next
```

```
PopClient1.Disconnect
```

Although this is very similar to the previous example that listed the available messages in the mailbox, there is an important difference. Whenever you plan on deleting messages from a POP3 mailbox, you should use the **LastMessage** property to determine what the message number is for the last available message in the mailbox, not the **MessageCount** property. Whenever a message is deleted from the POP3 mailbox, the **MessageCount** property value will decrease by one. This is done to reflect the fact that after message has been deleted, it can no longer be accessed. This is different than the IMAP4 protocol which merely flags a message for deletion and that message can still be accessed until the mailbox is expunged.

Internet Message Access Protocol

When you delete a message from a mailbox using the IMAP4 protocol, the message is simply flagged for deletion. Unlike the POP3 protocol, where the message can no longer be accessed, an IMAP4 client can retrieve messages that have been deleted until the mailbox has been expunged. Here is an example of how you could store all of the messages in a user's Inbox and then delete them:

```
Dim nMessageId As Long
Dim strSubject As String
Dim nError As Long

' Establish a connection to the mail server using the default port
' and the specified username and password
nError = ImapClient1.Connect(strHostName, , strUserName, strPassword)
If nError Then
    MsgBox ImapClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' Select the user's Inbox where new messages arrive
nError = ImapClient1.SelectMailbox("Inbox")
If nError Then
    MsgBox ImapClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' If the MessageCount property returns 0, then there are no
' messages in the mailbox
If ImapClient1.MessageCount = 0 Then
    MsgBox "There are no messages in this mailbox", vbInformation
    ImapClient1.Disconnect
    Exit Sub
End If

' Initialize the ProgressBar control
ProgressBar1.Value = 0
ProgressBar1.Max = ImapClient1.MessageCount

For nMessageId = 1 To ImapClient1.MessageCount
    ' Create a file name based on the message number
    strFileName = strFolder + Format(nMessageId, "00000000") + ".eml"

    ' Store the message in the specified file
    nError = ImapClient1.StoreMessage(nMessageId, strFileName)
    If nError = 0 Then
        ' If the message was stored successfully, then delete
        ' it from the mailbox
        nError = ImapClient1.DeleteMessage(nMessageId)
    End If
```

```
    ' If there was an error, warn the user
    If nError > 0 Then
        MsgBox ImapClient1.LastErrorString, vbExclamation
        Exit For
    End If

    ProgressBar1.Value = nMessageId
    DoEvents
Next

' Unselect the current mailbox, expunging the deleted messages
ImapClient1.UnselectMailbox True
ImapClient1.Disconnect
```

You'll notice that this code is very similar to the POP3 example, with two significant differences. The **SelectMailbox** method is used to explicitly select the user's Inbox, where new messages are stored until they are moved or deleted by the user. Unlike POP3 which only deals with new messages, IMAP4 is designed to manage multiple mailboxes, so you need to specify which mailbox you want to use. The mailbox "Inbox" is a special mailbox defined by the IMAP4 standard where all new messages are stored. In addition, the **UnselectMailbox** method is used to explicitly expunge the deleted messages from the mailbox. Note that it is possible to unselect a mailbox and leave the deleted messages intact until a later time.

Because messages are only flagged for deletion, it is possible to check for deleted messages in a mailbox by setting the **Message** property to the desired message number, and then checking the value of the **MessageFlags** property. If the **imapFlagDeleted** bit (a value of 512) has been set, then the message has been marked for deletion. For example:

```
For nMessageId = 1 To ImapClient1.MessageCount
    ImapClient1.Message = nMessageId
    If (ImapClient1.MessageFlags And imapFlagDeleted) Then
        MsgBox "Message " & nMessageId & " has been deleted", vbInformation
    End If
Next
```

It is also possible to undelete a message by calling the **UndeleteMessage** method. If you have multiple messages in a mailbox marked for deletion and you want to prevent all of them from being deleted, you can call the **ReselectMailbox** method which will reset the state of the current mailbox.

Terminal Services

SocketTools includes several components which can be used to establish character-based terminal sessions with a server, including the Telnet and Remote Command (RSH) controls. Typically these controls are used in one of two general ways:

- Establish an interactive session with the server, just as if a character-based terminal or console is being used. The Terminal Emulation control is used to display output, usually emulating an ANSI or DEC VT-220 terminal.

- Establish a connection with the server where keystrokes are sent to the server as though it is interacting with a user, however the output is parsed and presented to the user in a Windows application. In this case, the user never sees a standard character-based terminal window. Instead, the application drives the terminal session in the background.

Because the Terminal Emulation control is separate from the controls which provide the actual connection to the server, you can use it only when needed for an interactive session. The control can be used to emulate a standard ANSI console, a DEC VT-100 and DEC-VT220 terminal. It supports colors, character-based line drawing, scrollable regions, the ability to select and copy text to the clipboard and a number of other advanced features. For a list of the escape sequences supported by the control, refer to the Control Sequences section of the technical reference.

Telnet and Remote Login

To establish a terminal session with a server, you can either use the Telnet control or the Remote Command control. Telnet is a standard protocol which is used to provide basic terminal services, and is more widely supported than the Remote Login (rlogin) protocol. The following example demonstrates connecting to a Telnet server, listing the files in the user's current directory and capturing the output:

```
Dim strCommand As String
Dim strOutput As String
Dim strBuffer As String
Dim nResult As Long
Dim nError As Long

' Establish a connection with the server
nError = TelnetClient1.Connect(strHostName)
If nError Then
    MsgBox TelnetClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' Login to the server as the specified user
nError = TelnetClient1.Login(strUserName, strPassword)
If nError Then
    MsgBox TelnetClient1.LastErrorString, vbExclamation
    TelnetClient1.Disconnect
    Exit Sub
End If

' Issue a UNIX command to list the contents of the user's
' home directory and then logout
strCommand = "/bin/ls -l; exit" & vbCrLf
TelnetClient1.Write strCommand, Len(strCommand)

' Read the data returned by the server and collect it
' in the strOutput string
Do
    nResult = TelnetClient1.Read(strBuffer, 4096)
    If nResult > 1 Then strOutput = strOutput + strBuffer
Loop Until nResult < 1

' Disconnect from the server
TelnetClient1.Disconnect
```

After the connection has been established, the **Login** method is used to automatically login the user. It is important to note that this method expects that the server will respond with the standard Username: and Password: prompts that most UNIX and Windows based Telnet servers use. If the server uses a non-standard login sequence, then the client will need to use the **Search** method to search for the prompts and write code to respond to them. Note that many servers are configured to not permit the administrator (root) account to login using Telnet. If you are unable to login, check with your system administrator to determine if you have sufficient privileges to establish a telnet session from your local system.

An example using the Remote Command control to login to a server is very similar, however there are a few important differences:

```
Dim strCommand As String
Dim strOutput As String
Dim strBuffer As String
Dim nResult As Long
```

```
Dim nError As Long

' Login to the server as the specified user
nError = RshClient1.Login(strHostName, , strUserName)
If nError Then
    MsgBox RshClient1.LastErrorString, vbExclamation
    RshClient1.Disconnect
    Exit Sub
End If

' Issue a UNIX command to list the contents of the user's
' home directory and then logout
strCommand = "/bin/ls -l; exit" & vbCrLf
RshClient1.Write strCommand, Len(strCommand)

' Read the data returned by the server and collect it
' in the strOutput string
Do
    nResult = RshClient1.Read(strBuffer, 4096)
    If nResult > 1 Then strOutput = strOutput + strBuffer
Loop Until nResult < 1

' Disconnect from the server
RshClient1.Disconnect
```

Instead of a **Connect** method, the control uses the **Login** method to establish the connection and login the user. It is also important to note that a username is provided, but there is no password. This is because the rlogin protocol uses a concept called host equivalence. This means that the server must be specifically configured to allow the user to login from your local system. If the system administrator has not done this, then attempts to connect to the server will fail with the error that the connection has been aborted. If you need to use the rlogin protocol, most likely you will need to contact your system administrator to make the appropriate changes to permit access to the server.

Remote Command Execution

To execute a command on a server, you can Remote Command control. The following example demonstrates connecting to a UNIX server, listing the files in the user's current directory and capturing the output:

```
Dim strCommand As String
Dim strBuffer As String
Dim strOutput As String
Dim nResult As Long, nError As Long

' Execute the command on the server
nError = RshClient1.Execute(strHostName, rshPortExec, strUserName, strPassword,
strCommand)
If nError > 0 Then
    MsgBox RshClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' Read the output from the command and store it in a
' string buffer
Do
    nResult = RshClient1.Read(strBuffer, 1024)
    If nResult > 1 Then strOutput = strOutput + strBuffer
Loop Until nResult < 1

' Disconnect from the server
RshClient1.Disconnect
```

The Execute method establishes the connection to the server, authenticates the user and executes the command. The output from the command is read using the Read method. Note that this method should only be used if the command is not interactive and expect character-mode input. For example, the Execute method should not be used with editors such as Emacs. For interactive sessions or the ability to execute multiple commands in a single client session, the Telnet or Rlogin protocols should be used instead.

Domain Name Service Control

Resolve domain names into Internet addresses and return information about a remote host, such as the servers that are responsible for accepting mail for the domain.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	DnsClientCtl.DnsClient
File Name	CSDNSX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.DnsClient.11
ClassID	17CF34F6-C9D1-41B1-8FA7-647148185A2E
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 1034

Overview

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

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

Domain Name Service Properties

Property	Description
HostAddress	Gets and sets the IP address of the remote host
HostAlias	Returns the aliases defined for the current hostname
HostAliases	Return the number of aliases for the specified host name
HostFile	Gets and sets the name of an alternate host file
HostInfo	Returns information about the host system
HostName	Gets and sets the name of the remote host
HostProtocol	Set the protocol to return service information for the specified host
HostServices	Return the well-known services available for the specified host
IsBlocked	Determine if the control is blocked performing an operation
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 to occur
LocalAddress	Return the IP address of the local host
LocalDomain	Gets and sets the domain name for the local system
LocalName	Return the name of the local host
MailExchange	Return the name of the mail exchange host for the specified domain
MailExchanges	Return the number of mail exchange records for the current host
NameServer	Gets and sets the IP address of a nameserver
RemotePort	Gets and sets the port number for a remote connection
Retry	Set the number of times the control attempts to resolve a hostname
ServerAddress	Return the address of the nameserver that resolved the query
ThrowError	Enable or disable error handling by the container of the control
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 object

HostAddress Property

Gets and sets the IP address of the remote host.

Syntax

object.**HostAddress** [= *ipaddress*]

Remarks

Setting the **HostAddress** property causes the control to submit a reverse query to the nameservers that you have specified. If a reverse entry is found for the IP address, the **HostName** property is changed to that host name.

Note that reverse DNS entries may not be available for many systems.

Data Type

String

See Also

HostName Property, NameServer Property

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 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. The property array is zero based, meaning that the first index value is zero.

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
DnsClient1.HostName = Trim(Text1.Text)

For nIndex = 0 To DnsClient1.HostAliases - 1
     List1.AddItem DnsClient1.HostAlias(nIndex)
     nIndex = nIndex + 1
Loop
```

See Also

HostAddress Property, HostAliases Property, HostName Property

HostAliases Property

Return the number of aliases for the specified host name.

Syntax

object.**HostAliases**

Remarks

The **HostAliases** property returns the number of aliases for the host specified by the **HostName** property. If the specified host name cannot be resolved, this property will return a value of zero.

Data Type

Integer (Int32)

Example

The following example places the all of the aliases for a specific host into a listbox:

```
Dim nIndex As Integer

List1.Clear
DnsClient1.HostName = Trim(Text1.Text)

For nIndex = 0 To DnsClient1.HostAliases - 1
     List1.AddItem DnsClient1.HostAlias(nIndex)
     nIndex = nIndex + 1
Loop
```

See Also

HostAddress Property, HostAlias 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:

```
address 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.

Data Type

String

See Also

HostAddress Property, HostName Property, LocalName Property

HostInfo Property

Returns information about the host system.

Syntax

object.**HostInfo**

Remarks

The **HostInfo** property returns a string that includes the machine type and operating system for the host. This information corresponds to it's HINFO record in the database.

Data Type

String

See Also

HostAddress Property, HostName Property, HostProtocol Property, HostServices Property

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 obtain information on. Setting this property causes a query to be submitted to the nameservers that you have specified. If the host name can be resolved, the **HostAddress** property is set to the IP address of the host, in dot-notation.

Data Type

String

See Also

HostAddress Property, NameServer Property

HostProtocol Property

Set the protocol to return service information for the specified host.

Syntax

object.**HostProtocol** [= *protocol*]

Remarks

The **HostProtocol** property determines the protocol for which service information is returned, and should be set to one of the following values:

Value	Description
dnsProtocolTCP	Specifies services using the Transmission Control Protocol (TCP).
dnsProtocolUDP	Specifies services using the User Data Protocol (UDP).

Data Type

Integer (Int32)

See Also

HostServices Property

HostServices Property

Return the well-known services available for the specified host.

Syntax

object.**HostServices**

Remarks

The **HostServices** property returns a string that contains a list of well-known services for the specified host. This corresponds to the WKS entry in the nameserver's database. The services returned depend on the protocol specified in the **HostProtocol** property.

Data Type

String

See Also

HostProtocol Property

IsBlocked Property

Determine 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.

Note that this property will return True if there is ***any*** blocking operation being performed by the application, regardless if the specified control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

LastError Property

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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError 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 (four numbers separated by periods).

Data Type

String

See Also

HostAddress Property, LocalName Property

LocalDomain Property

Gets and sets the domain name for the local system.

Syntax

object.**LocalDomain** [= *domain*]

Remarks

The **LocalDomain** property is used to set the domain name for the local host. The local domain name is used when the name assigned to **HostName** property does not specify a domain (in other words, does not have a dot in the name). In that case, the value of the **LocalDomain** property is appended to the hostname.

If a domain name has been specified for the local system, the **LocalDomain** property is set to that value by default.

Data Type

String

See Also

HostAddress Property, HostName Property

LocalName Property

Return the name of the local host.

Syntax

object.**LocalName**

Remarks

The **LocalName** read-only property returns the name of the local host. The name that is returned depends on the configuration of the TCP/IP software.

Data Type

String

See Also

HostName Property, LocalAddress Property

MailExchange Property

Return the name of the mail exchange host for the specified domain.

Syntax

object.**MailExchange**(*Index*)

Remarks

The **MailExchange** property array returns the host name of the systems designated as the mail exchangers for the current domain. The mail exchange hosts are returned sorted in priority order, with the higher priority mail servers being listed first. The property array is zero based, which means that the first index value is zero. The **HostName** property must be set to the domain name that you want to obtain the mail exchange records for.

This property array is commonly used to determine which system is responsible for forwarding mail within a domain. For example, if a mail message is addressed to the user **someone@example.com**, you can determine the name of the server or servers responsible for accepting mail for that user by setting the value of the HostName property to **example.com** and then checking the **MailExchange** property array. Note that it is possible that a domain will not have any mail exchange (MX) records, in which case you should attempt to to connect directly to a mail server running on the host specified in the domain name portion of the address.

Data Type

String

Example

The following example places the all of the mail exchanges for a specific host into a listbox:

```
Dim nIndex As Integer

List1.Clear
DnsClient1.HostName = Trim(Text1.Text)

For nIndex = 0 To DnsClient1.MailExchanges - 1
     List1.AddItem DnsClient1.MailExchange(nIndex)
     nIndex = nIndex + 1
Loop
```

See Also

HostName Property, HostAddress Property, NameServer Property, MailExchanges Property

MailExchanges Property

Return the number of mail exchange records for the current host.

Syntax

object.**MailExchanges**

Remarks

The **MailExchanges** property returns the number of mail exchange (MX) records for the current host specified by the **HostName** property. This property can be used in conjunction with the **MailExchange** property to enumerate the servers responsible for accepting mail for a given domain.

Data Type

Integer (Int32)

Example

The following example places the all of the mail exchange records for a specific host into a listbox:

```
Dim nIndex As Integer

List1.Clear
DnsClient1.HostName = Trim(Text1.Text)

For nIndex = 0 To DnsClient1.MailExchanges - 1
     List1.AddItem DnsClient1.MailExchange(nIndex)
     nIndex = nIndex + 1
Loop
```

See Also

HostName Property, HostAddress Property, NameServer Property, MailExchange Property

NameServer Property

Gets and sets the IP address of a nameserver.

Syntax

object.**NameServer(***Index***)** [= *address*]

Remarks

The **NameServer** property array is used to specify one or more nameservers used to resolve hostnames and addresses. The address value must be an IP address in dot notation. The Index argument specifies which nameserver to set or return a value for. There may be up to 3 nameservers defined for any single instance of the control.

Data Type

String

See Also

HostAddress Property, HostName Property, Retry Property

RemotePort Property

Gets and sets the port number for a remote connection.

Syntax

object.**RemotePort** [= *port*]

Remarks

The **RemotePort** property is used to set the port number that the control will use to establish a connection with the server.

Data Type

Integer (Int32)

See Also

HostName Property

Retry Property

Set the number of times the control attempts to resolve a hostname.

Syntax

object.**Retry** [= *count*]

Remarks

The **Retry** property specifies the number of times, per nameserver, that the control attempts to resolve a hostname or address. If attempts to query a nameserver fail, the control waits a period of time and then resubmits the query. As the number of retries increase, the longer the period of time the control waits to receive a response before attempting the query using another nameserver.

The default number of retries is four, with a minimum value of one and a maximum value of eight.

Data Type

Integer (Int32)

See Also

NameServer Property

ServerAddress Property

Return the address of the nameserver that resolved the query.

Syntax

object.**ServerAddress**

Remarks

The **ServerAddress** property returns the IP address (in dot notation) of the nameserver that resolved the previous query.

Data Type

String

See Also

HostAddress Property, HostName Property, HostProtocol Property

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
DnsClient1.ThrowError = False
nError = DnsClient1.Resolve(strHostName, strHostAddress)

If nError > 0 Then
    MsgBox DnsClient1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

DnsClient1.ThrowError = True
DnsClient1.Resolve strHostName, strHostAddress

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error.

Note that the **Timeout** property also determines the amount of time the control will spend attempting to connect to a remote host. If a connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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 written to the trace file, including information about successful calls made to the networking library. This is the default value.
dnsTraceError	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.
dnsTraceWarning	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.
dnsTraceHexDump	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.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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 object.

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

Domain Name Service Methods

Method	Description
Cancel	Cancels the current blocking network operation
Initialize	Initialize the control and validate the runtime license key
MatchHost	Match a host name against of list of addresses including wildcards
Query	Perform a general nameserver query for a specific type of record
Reset	Reset the internal state of the control
Resolve	Resolves a host name to a host IP address
Uninitialize	Uninitialize the control and release any system resources that were allocated

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

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 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 dnsClient = CreateObject("SocketTools.DnsClient.11")

nError = dnsClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize Method

MatchHost Method

Match a host name against one more strings that may contain wildcards.

Syntax

object.**MatchHost**(*HostName*, *HostMask*, [*Resolve*])

Parameters

HostName

A string value which specifies the host name or IP address to match against the host mask string.

HostMask

A string value 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.

Resolve

An optional boolean value which determines if the host name or IP address should be resolved when matching the host against the mask string. If this parameter is True, 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 False, then the match is made only against the host name string provided. If this argument is omitted, the default value is True.

Return Value

A value of True is returned if the host name or IP address matched the host mask. Otherwise, a value of False is returned which indicates that there was no match.

Remarks

The **MatchHost** 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 function would return True, indicating the host name matched the mask. However, if the mask string was "*.net" then the function would return False, 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.

See Also

HostAddress Property, HostFile Property, HostName Property

Query Method

Perform a general nameserver query for a specific type of record.

Syntax

object.**Query**(*RecordName*, *RecordType*, *RecordData*, [*Reserved*])

Parameters

RecordName

A string value that specifies the name of the record that is to be retrieved. Typically this is the name of a given host for which you wish to obtain information.

RecordType

An integer value that specifies the type of record that is to be retrieved. This parameter should be set to one of the following values:

Value	Description
dnsRecordNone	No record type
dnsRecordAddress	Host address
dnsRecordNS	Authoritative nameserver
dnsRecordCName	Canonical name (alias)
dnsRecordSOA	Start of Authority
dnsRecordWKS	Well known services
dnsRecordPTR	Domain name
dnsRecordHInfo	Host information
dnsRecordMInfo	Mailbox information
dnsRecordMX	Mail exchange host
dnsRecordTXT	Text strings
dnsRecordLoc	Location information
dnsRecordUInfo	User information
dnsRecordUid	User ID
dnsRecordGid	Group ID

RecordData

A string variable that is set to the data returned as a result of the query. If no data was returned, this argument will be set to an empty string. This parameter must be passed by reference.

Reserved

An optional parameter that is reserved for future use. This parameter should be omitted or passed as an empty variant.

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 **Query** method performs a general nameserver query for a given record based on the name and type. This method will not use a local host file when performing host name or IP address lookups.

The **dnsRecordAddress** record type is used to resolve a host name into an IP address, and is the most common type of nameserver query that is performed. This is the type of query that is used when you set the **HostName** property and then read the **HostAddress** property to obtain it's IP address. If the host name has both an IPv4 and IPv6 address, this method will return the IPv4 address by default for compatibility with existing applications. It will only return an IPv6 address if the host has no IPv4 address assigned to it.

The **dnsRecordPTR** record type is used to resolve an IP address into a host name, and is also referred to as a reverse DNS lookup. The ***RecordName*** argument should be set to the IP address of the system, and the ***RecordData*** argument will contain its fully qualified domain name when the method returns. Note that this requires that a PTR record actually exists for the given address, which may not be the case. This is the same type of query that is performed when you set the **HostAddress** property and then read the **HostName** property to determine the host name.

See Also

HostAddress Property, HostName Property, MailExchange Property, Resolve 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, 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*, [*Reserved*])

Parameters

HostName

A string value that specifies the host name to resolve.

IpAddress

A string variable that will contain the IP address for the specified host name when the method returns. This parameter must be passed by reference.

Reserved

An optional parameter that is reserved for future use. This parameter should be omitted or passed as an empty variant.

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 convert a host name into an IP address. Note that unlike the **Query** method, this method will use the local host file when resolving the host name. If the host name has both an IPv4 and IPv6 address, this method will return the IPv4 address by default for compatibility with existing applications. It will only return an IPv6 address if the host has no IPv4 address assigned to it.

See Also

HostAddress Property, HostFile Property, HostName Property, Query 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. 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

Domain Name Service Events

Event	Description
OnError	This event is generated when a control error occurs

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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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

SocketTools Control Error Codes

Value	Constant	Description
10001	stErrorNotHandleOwner	Handle not owned by the current thread
10002	stErrorFileNotFound	The specified file or directory does not exist
10003	stErrorFileNotCreated	The specified file could not be created
10004	stErrorOperationCanceled	The blocking operation has been canceled
10005	stErrorInvalidFileType	The specified file is a block or character device, not a regular file
10006	stErrorInvalidDevice	The specified device or address does not exist
10007	stErrorTooManyParameters	The maximum number of function parameters has been exceeded
10008	stErrorInvalidFileName	The specified file name contains invalid characters or is too long
10009	stErrorInvalidFileHandle	Invalid file handle passed to function
10010	stErrorFileReadFailed	Unable to read data from the specified file
10011	stErrorFileWriteFailed	Unable to write data to the specified file
10012	stErrorOutOfMemory	Out of memory
10013	stErrorAccessDenied	Access denied
10014	stErrorInvalidParameter	Invalid argument passed to function
10015	stErrorClipboardUnavailable	The system clipboard is currently unavailable
10016	stErrorClipboardEmpty	The system clipboard is empty or does not contain any text data
10017	stErrorFileEmpty	The specified file does not contain any data
10018	stErrorFileExists	The specified file already exists
10019	stErrorEndOfFile	End of file
10020	stErrorDeviceNotFound	The specified device could not be found
10021	stErrorDirectoryNotFound	The specified directory could not be found
10022	stErrorInvalidBuffer	Invalid memory address passed to function
10024	stErrorNoHandles	No more handles available to this process
10035	stErrorOperationWouldBlock	The specified operation would block the current thread
10036	stErrorOperationInProgress	A blocking operation is currently in progress
10037	stErrorAlreadyInProgress	The specified operation is already in progress
10038	stErrorInvalidHandle	Invalid handle passed to function
10039	stErrorInvalidAddress	Invalid network address specified
10040	stErrorInvalidSize	Datagram is too large to fit in specified buffer

10041	stErrorInvalidProtocol	Invalid network protocol specified
10042	stErrorProtocolNotAvailable	The specified network protocol is not available
10043	stErrorProtocolNotSupported	The specified protocol is not supported
10044	stErrorSocketNotSupported	The specified socket type is not supported
10045	stErrorInvalidOption	The specified option is invalid
10046	stErrorProtocolFamily	The specified protocol family is not supported
10047	stErrorProtocolAddress	The specified address is invalid for this protocol family
10048	stErrorAddressInUse	The specified address is in use by another process
10049	stErrorAddressUnavailable	The specified address cannot be assigned
10050	stErrorNetworkUnavailable	The networking subsystem is unavailable
10051	stErrorNetworkUnreachable	The specified network is unreachable
10052	stErrorNetworkReset	Network dropped connection on reset
10053	stErrorConnectionAborted	Connection was aborted due to timeout or other failure
10054	stErrorConnectionReset	Connection was reset by remote network
10055	stErrorOutOfBuffers	No buffer space is available
10056	stErrorAlreadyConnected	Connection already established with remote host
10057	stErrorNotConnected	No connection established with remote host
10058	stErrorConnectionShutdown	Unable to send or receive data after connection shutdown
10060	stErrorOperationTimeout	The specified operation has timed out
10061	stErrorConnectionRefused	The connection has been refused by the remote host
10064	stErrorHostUnavailable	The specified host is unavailable
10065	stErrorHostUnreachable	The specified host is unreachable
10067	stErrorTooManyProcesses	Too many processes are using the networking subsystem
10069	stErrorTooManyThreads	Too many threads have been created by the current process
10070	stErrorTooManySessions	Too many client sessions have been created by the current process
10082	stErrorInternalFailure	An unexpected internal error has occurred
10091	stErrorNetworkNotReady	Network subsystem is not ready for communication
10092	stErrorInvalidVersion	This version of the operating system is not supported
10093	stErrorNetworkNotInitialized	The networking subsystem has not been initialized
10101	stErrorRemoteShutdown	The remote host has initiated a graceful shutdown sequence
11001	stErrorInvalidHostName	The specified hostname is invalid or could not be resolved
11002	stErrorHostNameNotFound	The specified hostname could not be found
11003	stErrorHostNameRefused	Unable to resolve hostname, request refused
11004	stErrorHostNameNotResolved	Unable to resolve hostname, no address for specified host

12001	stErrorInvalidLicense	The license for this product is invalid
12002	stErrorProductNotLicensed	This product is not licensed to perform this operation
12003	stErrorNotImplemented	This function has not been implemented on this platform
12004	stErrorUnknownLocalHost	Unable to determine local host name
12005	stErrorInvalidHostAddress	Invalid host address specified
12006	stErrorInvalidServicePort	Invalid service port number specified
12007	stErrorInvalidServiceName	Invalid or unknown service name specified
12008	stErrorInvalidEventId	Invalid event identifier specified
12009	stErrorOperationNotBlocking	No blocking operation in progress on this socket
12101	stErrorSecurityNotInitialized	Unable to initialize security interface for this process
12102	stErrorSecurityContext	Unable to establish security context for this session
12103	stErrorSecurityCredentials	Unable to open client certificate store or establish client credentials
12104	stErrorSecurityCertificate	Unable to validate the certificate chain for this session
12105	stErrorSecurityDecryption	Unable to decrypt data stream
12106	stErrorSecurityEncryption	Unable to encrypt data stream
12201	stErrorOperationNotSupported	The specified operation is not supported
12202	stErrorInvalidProtocolVersion	Invalid application protocol version specified
12203	stErrorNoServerResponse	No data returned from server
12204	stErrorInvalidServerResponse	Invalid data returned from server
12205	stErrorUnexpectedServerResponse	Unexpected response code returned from server
12206	stErrorServerTransactionFailed	Server transaction failed
12207	stErrorServiceUnavailable	The service is currently unavailable
12208	stErrorServiceNotReady	The service is not ready, try again later
12209	stErrorServerResyncFailed	Unable to resynchronize with server
12210	stErrorInvalidProxyType	Invalid proxy server type specified
12211	stErrorProxyRequired	Resource must be accessed through specified proxy
12212	stErrorInvalidProxyLogin	Unable to login to proxy server using specified credentials
12213	stErrorProxyResyncFailed	Unable to resynchronize with proxy server
12214	stErrorInvalidCommand	Invalid command specified
12215	stErrorInvalidCommandParameter	Invalid command parameter specified
12216	stErrorInvalidCommandSequence	Invalid command sequence specified
12217	stErrorCommandNotImplemented	Specified command not implemented on this server
12218	stErrorCommandNotAuthorized	Specified command not authorized for the current user
12219	stErrorCommandAborted	Specified command was aborted by the remote host

12220	stErrorOptionNotSupported	The specified option is not supported on this server
12221	stErrorRequestNotCompleted	The current client request has not been completed
12222	stErrorInvalidUsername	The specified username is invalid
12223	stErrorInvalidPassword	The specified password is invalid
12224	stErrorInvalidAccount	The specified account name is invalid
12225	stErrorAccountRequired	Account name has not been specified
12226	stErrorInvalidAuthenticationType	Invalid authentication protocol specified
12227	stErrorAuthenticationRequired	User authentication is required
12228	stErrorProxyAuthenticationRequired	Proxy authentication required
12229	stErrorAlreadyAuthenticated	User has already been authenticated
12230	stErrorAuthenticationFailed	Unable to authenticate the specified user
12251	stErrorNetworkAdapter	Unable to determine network adapter configuration
12252	stErrorInvalidRecordType	Invalid record type specified
12253	stErrorInvalidRecordName	Invalid record name specified
12254	stErrorInvalidRecordData	Invalid record data specified
12255	stErrorConnectionOpen	Data connection already established
12256	stErrorConnectionClosed	Server closed data connection
12257	stErrorConnectionPassive	Data connection is passive
12258	stErrorConnectionFailed	Unable to open data connection to server
12259	stErrorInvalidSecurityLevel	Data connection cannot be opened with this security setting
12260	stErrorCachedTlsRequired	Data connection requires cached TLS session
12261	stErrorDataReadOnly	Data connection is read-only
12262	stErrorDataWriteOnly	Data connection is write-only
12263	stErrorEndOfData	End of data
12264	stErrorRemoteFileUnavailable	Remote file is unavailable
12265	stErrorInsufficientStorage	Insufficient storage on server
12266	stErrorStorageAllocation	File exceeded storage allocation on server
12267	stErrorDirectoryExists	The specified directory already exists
12268	stErrorDirectoryEmpty	No files returned by the server for the specified directory
12269	stErrorEndOfDirectory	End of directory listing
12270	stErrorUnknownDirectoryFormat	Unknown directory format
12271	stErrorInvalidResource	Invalid resource name specified
12272	stErrorResourceRedirected	The specified resource has been redirected
12273	stErrorResourceRestricted	Access to this resource has been restricted
12274	stErrorResourceNotModified	The specified resource has not been modified

12275	stErrorResourceNotFound	The specified resource cannot be found
12276	stErrorResourceConflict	Request could not be completed due to the current state of the resource
12277	stErrorResourceRemoved	The specified resource has been permanently removed from this server
12278	stErrorContentLengthRequired	Request must include the content length
12279	stErrorRequestPrecondition	Request could not be completed due to server precondition
12280	stErrorUnsupportedMediaType	Request specified an unsupported media type
12281	stErrorInvalidContentRange	Content range specified for this resource is invalid
12282	stErrorInvalidMessagePart	Message is not multipart or an invalid message part was specified
12283	stErrorInvalidMessageHeader	The specified message header is invalid or has not been defined
12284	stErrorInvalidMessageBoundary	The multipart message boundary has not been defined
12285	stErrorNoFileAttachment	The current message part does not contain a file attachment
12286	stErrorUnknownFileType	The specified file type could not be determined
12287	stErrorDataNotEncoded	The specified data block could not be encoded
12288	stErrorDataNotDecoded	The specified data block could not be decoded
12289	stErrorFileNotEncoded	The specified file could not be encoded
12290	stErrorFileNotDecoded	The specified file could not be decoded
12291	stErrorNoMessageText	No message text
12292	stErrorInvalidCharacterSet	Invalid character set specified
12293	stErrorInvalidEncodingType	Invalid encoding type specified
12294	stErrorInvalidMessageNumber	Invalid message number specified
12295	stErrorNoReturnAddress	No valid return address specified
12296	stErrorNoValidRecipients	No valid recipients specified
12297	stErrorInvalidRecipient	The specified recipient address is invalid
12298	stErrorRelayNotAuthorized	The specified domain is invalid or server will not relay messages
12299	stErrorMailboxUnavailable	Specified mailbox is currently unavailable
12300	stErrorMailboxReadonly	The selected mailbox cannot be modified
12301	stErrorMailboxNotSelected	No mailbox has been selected
12302	stErrorInvalidMailbox	Specified mailbox is invalid
12303	stErrorInvalidDomain	The specified domain name is invalid or not recognized
12304	stErrorInvalidSender	The specified sender address is invalid or not recognized
12305	stErrorMessageNotDelivered	Message not delivered to any of the specified recipients

12306	stErrorEndOfMessageData	No more message data available to be read
12307	stErrorInvalidMessageSize	The specified message size is invalid
12308	stErrorMessageNotCreated	The message could not be created in the specified mailbox
12309	stErrorNoMoreMailboxes	No more mailboxes exist on this server
12310	stErrorInvalidEmulationType	The specified terminal emulation type is invalid
12311	stErrorInvalidFontHandle	The specified font handle is invalid
12312	stErrorInvalidFontName	The specified font name is invalid or unavailable
12313	stErrorInvalidPacketSize	The specified packet size is invalid
12314	stErrorInvalidPacketData	The specified packet data is invalid
12315	stErrorInvalidPacketId	The unique packet identifier is invalid
12316	stErrorPacketTtlExpired	The specified packet time-to-live period has expired
12317	stErrorInvalidNewsgroup	Invalid newsgroup specified
12318	stErrorNoNewsgroupSelected	No newsgroup selected
12319	stErrorEmptyNewsgroup	No articles in specified newsgroup
12320	stErrorInvalidArticle	Invalid article number specified
12321	stErrorNoArticleSelected	No article selected in the current newsgroup
12322	stErrorFirstArticle	First article in current newsgroup
12323	stErrorLastArticle	Last article in current newsgroup
12324	stErrorArticleExists	Unable to transfer article, article already exists
12325	stErrorArticleRejected	Unable to transfer article, article rejected
12326	stErrorArticleTransferFailed	Article transfer failed
12327	stErrorArticlePostingDenied	Posting is not permitted on this server
12328	stErrorArticlePostingFailed	Posting is not permitted on this server
12329	stErrorInvalidDateFormat	The specified date format is not recognized
12330	stErrorFeatureNotSupported	The specified feature is not supported on this server
12331	stErrorInvalidFormHandle	The specified form handle is invalid or a form has not been created
12332	stErrorInvalidFormAction	The specified form action is invalid or has not been specified
12333	stErrorInvalidFormMethod	The specified form method is invalid or not supported
12334	stErrorInvalidFormType	The specified form type is invalid or not supported
12335	stErrorInvalidFormField	The specified form field name is invalid or does not exist
12336	stErrorEmptyForm	The specified form does not contain any field values
12337	stErrorMaximumConnections	The maximum number of client connections exceeded
12338	stErrorThreadCreationFailed	Unable to create a new thread for the current process
12339	stErrorInvalidThreadHandle	The specified thread handle is no longer valid

12340	stErrorThreadTerminated	The specified thread has been terminated
12341	stErrorThreadDeadlock	The operation would result in the current thread becoming deadlocked
12342	stErrorInvalidClientMoniker	The specified moniker is not associated with any client session
12343	stErrorClientMonikerExists	The specified moniker has been assigned to another client session
12344	stErrorServerInactive	The specified server is not listening for client connections
12345	stErrorServerSuspended	The specified server is suspended and not accepting client connections
12346	stErrorNoMessageStore	No message store has been specified
12347	stErrorMessageStoreChanged	The message store has changed since it was last accessed
12348	stErrorMessageNotFound	No message was found that matches the specified criteria
12349	stErrorMessageDeleted	The specified message has been deleted
12350	stErrorFileChecksumMismatch	The specified file checksums do not match
12351	stErrorFileSizeMismatch	The specified file sizes do not match
12352	stErrorInvalidFeedUrl	The news feed URL is invalid or specifies an unsupported protocol
12353	stErrorInvalidFeedFormat	The internal format of the news feed is invalid
12354	stErrorInvalidFeedVersion	This version of the news feed is not supported
12355	stErrorChannelEmpty	There are no valid items found in this news feed
12356	stErrorInvalidItemNumber	The specified channel item identifier is invalid
12357	stErrorItemNotFound	The specified channel item could not be found
12358	stErrorItemEmpty	The specified channel item does not contain any data
12359	stErrorInvalidItemProperty	The specified item property name is invalid
12360	stErrorItemPropertyNotFound	The specified item property has not been defined
12361	stErrorInvalidChannelTitle	The channel title is invalid or has not been defined
12362	stErrorInvalidChannelLink	The channel hyperlink is invalid or has not been defined
12363	stErrorInvalidChannelDescription	The channel description is invalid or has not been defined
12364	stErrorInvalidItemText	The description for an item is invalid or has not been defined
12365	stErrorInvalidItemLink	The hyperlink for an item is invalid or has not been defined
12366	stErrorInvalidServiceType	The specified service type is invalid
12367	stErrorServiceSupended	Access to the specified service has been suspended
12368	stErrorServiceRestricted	Access to the specified service has been restricted
12369	stErrorInvalidProviderName	The specified provider name is invalid or unknown
12370	stErrorInvalidPhoneNumber	The specified phone number is invalid or not supported in this region

12371	stErrorGatewayNotFound	A message gateway cannot be found for the specified provider
12372	stErrorMessageTooLong	The message exceeds the maximum number of characters permitted
12373	stErrorInvalidProviderData	The request returned invalid or incomplete service provider data
12374	stErrorInvalidGatewayData	The request returned invalid or incomplete message gateway data
12375	stErrorMultipleProviders	The request has returned multiple service providers
12376	stErrorProviderNotFound	The specified service provider could not be found
12377	stErrorInvalidMessageService	The specified message is not supported with this service type
12378	stErrorInvalidMessageFormat	The specified message format is invalid
12379	stErrorInvalidConfiguration	The specified configuration options are invalid
12380	stErrorServerActive	The requested action is not permitted while the server is active
12381	stErrorServerPortBound	Unable to obtain exclusive use of the specified local port
12382	stErrorInvalidClientSession	The specified client identifier is invalid for this session
12383	stErrorClientNotIdentified	The specified client has not provided user credentials
12384	stErrorInvalidClientState	The requested action cannot be performed at this time
12385	stErrorInvalidResultCode	The specified result code is not valid for this protocol
12386	stErrorCommandRequired	The specified command is required and cannot be disabled
12387	stErrorCommandDisabled	The specified command has been disabled
12388	stErrorCommandSequence	The command cannot be processed at this time
12389	stErrorCommandCompleted	The previous command has completed
12390	stErrorInvalidProgramName	The specified program name is invalid or unrecognized
12391	stErrorInvalidRequestHeader	The request header contains one or more invalid values
12392	stErrorInvalidVirtualHost	The specified virtual host name is invalid
12393	stErrorVirtualHostNotFound	The specified virtual host does not exist
12394	stErrorTooManyVirtualHosts	Too many virtual hosts created for this server
12395	stErrorInvalidVirtualPath	The specified virtual path name is invalid
12396	stErrorVirtualPathNotFound	The specified virtual path does not exist
12397	stErrorTooManyVirtualPaths	Too many virtual paths created for this server
12398	stErrorInvalidTask	The asynchronous task identifier is invalid
12399	stErrorTaskFinished	The asynchronous task has not finished
12400	stErrorTaskQueued	The asynchronous task has been queued
12401	stErrorTaskSuspended	The asynchronous task has been suspended
12402	stErrorTaskFinished	The asynchronous task has finished

12403	stErrorInvalidAccountUuid	The application account identifier is invalid
12404	stErrorInvalidAccountId	The product identifier identifier is invalid
12405	stErrorInvalidProductId	TODO
12406	stErrorInvalidSerialNumber	The product serial number is invalid
12407	stErrorInvalidAppId	The application identifier is invalid
12408	stErrorInvalidApiKey	The application key is invalid
12409	stErrorAccountExists	The application account identifier already exists
12410	stErrorAccountNotCreated	The application account identifier was not created
12411	stErrorAccountNotFound	The application account identifier was not found
12412	stErrorAccountNotExpired	Access to this account has not expired
12413	stErrorAccountNotUpdated	The application account could not be updated
12414	stErrorAccountExpired	Access to this account has expired
12415	stErrorAccountRevoked	Access to this account has been revoked
12416	stErrorApiKeyNotCreated	The application key could not be created
12417	stErrorApiKeyNotFound	The application key could not be found
12418	stErrorApiKeyNotExpired	The application key has not expired
12419	stErrorApiKeyNotUnique	The application key identifier is not unique
12420	stErrorApiKeyNotUpdated	The application key could not be updated
12421	stErrorApiKeyNotDeleted	The application key could not be deleted
12422	stErrorApiKeyExists	The application key already exists
12423	stErrorApiKeyExpired	The application key has expired and must be refreshed
12424	stErrorApiKeyRevoked	TODO
12425	stErrorApiKeyAppId	The application key has been revoked
12426	stErrorInvalidToken	The application was not found or was not specified
12427	stErrorTokenNotCreated	The access token could not be created
12428	stErrorTokenNotFound	The access token could not be found
12429	stErrorTokenNotExpired	The access token has not expired
12430	stErrorTokenNotUpdated	The access token was not updated
12431	stErrorTokenNotDeleted	The access token could not be deleted
12432	stErrorTokenExpired	The access token has expired and must be refreshed
12433	stErrorTokenRevoked	The access token has been revoked
12434	stErrorNoApiKeysFound	No application keys found for this account
12435	stErrorNoTokensFound	No access tokens found for this application key
12436	stErrorNoTokensRevoked	No access tokens have been revoked
12437	stErrorInvalidStorageObject	Invalid storage object identifier

12438	stErrorStorageObjectReadOnly	The storage object is read-only
12439	stErrorStorageObjectExpired	Access to the storage object has expired
12440	stErrorStorageObjectSize	The storage object size exceeds storage limits
12441	stErrorStorageObjectDigest	The storage object digest is invalid or cannot be computed
12442	stErrorStorageObjectExists	A storage object with this label already exists
12443	stErrorStorageObjectModified	A storage object with this label has been modified
12444	stErrorStorageObjectNotOwner	The current user is not the storage object owner
12445	stErrorStorageObjectNotFound	The specified storage object does not exist
12446	stErrorStorageObjectNotCreated	The storage object was not created
12447	stErrorStorageObjectNotModified	The storage object was not modified
12448	stErrorStorageObjectNotRenamed	The storage object was not renamed
12449	stErrorStorageFolderEmpty	The storage folder does not contain any objects
12450	stErrorStorageAccountQuota	The storage account has exceeded its quota
12451	stErrorStorageAccountLimit	The storage account has exceeded its object limit
12452	stErrorInvalidStorageType	The specified storage type is invalid
12453	stErrorInvalidStorageProvider	The specified storage provider is not available
12454	stErrorInvalidStorageRegion	The specified storage region is not available
12455	stErrorInvalidStorageContainer	The storage container does not exist or cannot be accessed
12456	stErrorInvalidStorageLabel	The storage object label is invalid or undefined
12457	stErrorInvalidQueueHandle	The specified queue handle is invalid or the queue has been deleted
12458	stErrorInvalidQueueFile	The specified file identifier is not valid for this queue
12459	stErrorQueueRunning	The operation cannot be performed while the queue is running
12460	stErrorQueueStopped	The operation cannot be performed when the queue has stopped
12461	stErrorQueueEmpty	There are no files in the specified queue
12462	stErrorQueuePaused	The operation cannot be performed while the queue is paused
12463	stErrorQueueLocked	The operation cannot be performed while the queue is locked
12464	stErrorFileNotQueued	The specified file cannot be found in the queue
12465	stErrorEndOfQueue	There are no more files in the specified queue
12466	stErrorTooManyFiles	The maximum number of files have been queued for transfer
12467	stErrorNoQueuedTransfer	No queued file transfer is currently in progress
12468	stErrorIvalidX509Certificate	The specified X.509 format certificate is invalid
12469	stErrorInvalidPKCS12Certificiate	The specified PKCS 12 format certificate is invalid

12470	stErrorInvalidCipherSuite	The specified cipher suite is invalid or unavailable
12471	stErrorDeprecatedCipherSuite	The specified cipher suite is insecure and has been deprecated
12472	stErrorInvalidCertificateChain	The certificate chain could not be validated
12473	stErrorInvalidPrivateKey	The private key for the certificate is invalid
12474	stErrorInvalidApiSession	The application session identifier is invalid
12475	stErrorExpiredApiSession	The application session identifier has expired
12476	stErrorInvalidApiToken	The application token for this session is invalid
12477	stErrorExpiredApiToken	The application token for this session has expired
12478	stErrorInvalidApiAuthId	The authorization token for this session is invalid
12479	stErrorInvalidApiEndpoint	The endpoint for the specified request is invalid
12480	stErrorInvalidApiPayload	The data submitted with the specified request is invalid
12481	stErrorUnknownSessionOwner	The current session owner is unknown or no longer valid
12482	stErrorRevokedSessionAuth	The authorization token for this session has been revoked
12483	stErrorInvalidUrlScheme	The scheme for the specified URL is invalid or not supported
12484	stErrorInvalidUrlHost	The host name for the specified URL is invalid
12485	stErrorInvalidUrlPort	The port number for the specified URL is invalid
12486	stErrorInvalidUrlPath	The resource path for the specified URL is invalid
12487	stErrorInvalidContentType	The content type is invalid or not supported
12488	stErrorUnknownContentType	The content type cannot be determined
12489	stErrorInvalidCharset	The specified character set is invalid or not supported
12490	stErrorInvalidCodePage	The specified ANSI code page is invalid or not supported
12491	stErrorInvalidHeaderType	The specified header type is invalid or not supported
12492	stErrorInvalidHeaderName	The specified header name is invalid or not permitted
12493	stErrorInvalidHeaderValue	The specified header value is invalid or not permitted
12494	stErrorHeaderNotFound	The specified header value is undefined
12495	stErrorInvalidDigestType	The specified message digest type is invalid
12496	stErrorInvalidDigestValue	The specified message digest value is invalid
12497	stErrorFileDigestMismatch	The message digest does not match the specified file
12498	stErrorFileContentMismatch	The contents of the specified files are not identical
12499	stErrorFileTimeMismatch	The timestamps for the specified files are not identical

File Encoding Control

Encode and decode files using standard algorithms such as base64, uuencode and quoted-printable. The control can also be used to compress and expand files, as well as encrypt or decrypt file data using AES encryption.

Reference

- Properties
- Methods
- Events

Control Information

Object Name	FileEncoderCtl.FileEncoder
File Name	CSNCDX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.FileEncoder.11
ClassID	7C53A9D2-20C6-4BC3-BE1C-B7B4322D0BCE
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 1738, RFC 1951, RFC 2045

Overview

The File Encoding control provides functions for encoding and decoding binary files, typically attachments to email messages. The process of encoding converts the contents of a binary file to printable 7-bit ASCII text. Decoding reverses the process, converting a previously encoded text file back into a binary file.

There are two primary types of encoding methods used with various Internet applications: base64 and uucode. The base64 algorithm is most commonly used with email attachments, and is often referred to as MIME encoding since this is the encoding method specified in the MIME standards document. The uucode algorithm (so called because the programs to perform the encoding were called uuencode and uudecode) is often used when attaching binary files to Usenet newsgroup posts. The library also supports an alternate encoding format called yEnc which is also widely used to attach files to Usenet posts.

In addition to encoding and decoding data files, this control includes methods to compress and expand data, as well as encrypt and decrypt files using 256-bit AES encryption.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop

and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

File Encoding Control Properties

Property	Description
DecodedText	Set the current decoded text or return the decoded value of the **EncodedText** property
DecryptedText	Set the current decrypted text or return decrypted value of the **EncryptedText** property
EncodedText	Set the current encoded text or return the encoded value of the **DecodedText** property
Encoding	Gets and sets the current encoding method used by the component
EncryptedText	Set the current encrypted text or return the encrypted value of the **DecryptedText** property
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error to occur
Password	Gets and sets the password value used to encrypt and decrypt data
ThrowError	Enable or disable error handling by the container of the control
Version	Return the current version of the object

DecodedText Property

Set the value of the current decoded text string, or return the decoded value of an encoded string.

Syntax

object.**DecodedText** [= *value*]

Remarks

The **DecodedText** property is used to specify the decoded value of a string, or return the decoded value of a previously encoded string. If the property is set, the current plain (decoded) text is changed to that value and reading the **EncodedText** property will return the encoded string for this value. If the property is read, it will return the decoded value of the **EncodedText** property.

The control will use the value of the **Encoding** property to determine how the text should be decoded. Note that only base64 and quoted-printable encoding is supported when decoding a string using this property. To decode the contents of a file, it's recommended that you use the **DecodeFile** method.

This property only supports decoding text which was previously encoded using the ASCII or UTF-8 character set. If you need to decode text which used a different character set, you should use the **DecodeText** method in the Mail Message control. It is a component which is used to create and parse MIME formatted messages, and it is capable of decoding text that was encoded using a variety of different character sets.

Data Type

String

See Also

EncodedText Property, Encoding Property, DecodeFile Method, DecodeText Method

DecryptedText Property

Set the value of the current decrypted text string, or return the decrypted value of an encrypted string.

Syntax

object.**DecryptedText** [= *value*]

Remarks

The **DecryptedText** property is used to specify the current decrypted (plain text) value, or return the decrypted value of a previously encrypted string. If the property is set, the current text is changed to this value. Getting this property will decrypt the **EncryptedText** property value and return a copy of the decrypted string.

The value of the **Password** property will be used to generate the decryption key. If the **Password** property has not been set, or if it's an empty string, an default internal hash value is used to decrypt the data. Password values that exceed 215 characters will be truncated.

Due to how the decryption key is created internally, the control cannot be used to decrypt strings previously encrypted another third-party library or component. The encryption is performed using the 256-bit AES (Advanced Encryption Standard) algorithm, and the key is generated using an SHA-256 hash of the password value. It is not possible to recover previously encrypted text if the password value is unknown.

Data Type

String

See Also

EncryptedText Property, Password Property, DecryptData Method, DecryptFile Method, EncryptData Method

EncodedText Property

Set the value of the current encoded text string, or return the encoded value of a plain text string.

Syntax

object.**EncodedText** [= *value*]

Remarks

The **EncodedText** property is used to specify the encoded value of a string, or return the encoded value of a plain text (decoded) string. If the property is set, the current value will be changed to the encoded value, and the **DecodedText** property will return the plain text value of the encoded string. If the property is read, it will return the encoded value of the **DecodedText** property.

The control will use the value of the **Encoding** property to determine how the text should be encoded. Note that only base64 and quoted-printable encoding is supported when encoding a string using this property. To encode the contents of a file, it's recommended that you use the **EncodeFile** method.

This property only supports encoding text using the UTF-8 or ASCII character set. If you need to encode text using a different character set, you should use the **EncodeText** method in the Mail Message control. It is a component which is used to create and parse MIME formatted messages, and it is capable of encoding text using a variety of different character sets.

Data Type

String

See Also

DecodedText Property, Encoding Property, EncodeFile Method, EncodeText Method

Encoding Property

Gets and sets the current encoding method used by the control.

Syntax

object.**Encoding** [= *value*]

Remarks

The **Encoding** property determines how the specified file will be encoded or decoded by the control. The following encoding methods are supported:

Value	Description
fileEncodeDefault	Use the default encoding method. Currently this is the same specifying that the base64 algorithm should be used for encoding and decoding files.
fileEncodeBase64	Use the base64 algorithm for encoding and decoding files. This is the standard method for encoding files as outlined in the Multipurpose Internet Mail Extensions (MIME) protocol. This is the method used by most modern email client software.
fileEncodeQuoted	This encoding method is typically used for text messages that use characters beyond the standard ASCII character set, in the range of 128-255. This method, called quoted printable encoding, allows text messages to pass through mail systems that do not support characters with the high-bit set. Note that this method should not be used to encode binary files such as executables because the resulting output can be very large. For binary files, use the base64 algorithm instead.
fileEncodeUucode	Use the uuencode and uudecode algorithms for encoding and decoding files. This is a common encoding method used with UNIX systems and older email client software.
fileEncodeYencode	Use the yEnc algorithm for encoding the file. This is an encoding method that is commonly used when posting files to Usenet newsgroups.

The value of this property specifies the default encoding method for the **DecodeFile** and **EncodeFile** methods. Unless needed for a specific purpose, it is strongly recommended that binary files be encoded with the base64 algorithm for maximum compatibility. Note that it is not necessary to use this control to encode or decode file attachments with the Mail Message control, since it automatically handles encoding and decoding multipart messages.

The **Encoding** property also determines the encoding method that is used when the **DecodedText** and **EncodedText** properties are used to decode and encode text strings. The following values are supported:

Value	Description
dataEncodeDefault	Use the default encoding method. Currently this is the same specifying that the base64 algorithm should be used for encoding and decoding data.

dataEncodeBase64	Use the base64 algorithm for encoding and decoding data. This is the standard method for encoding data as outlined in the Multipurpose Internet Mail Extensions (MIME) protocol. This is the method used by most modern email client software.
dataEncodeQuoted	This encoding method is typically used for text messages that use characters beyond the standard ASCII character set, in the range of 128-255. This method, called quoted printable encoding, allows text messages to pass through mail systems that do not support characters with the high-bit set. Note that this method should not be used to encode binary data such as executables because the resulting output can be very large. For binary data, use the base64 algorithm instead.
dataEncodeURL	This encoding method is used with Uniform Resource Locators (URLs) to convert certain reserved characters to ensure that the URL is processed correctly by a web server. Numbers and letters are unchanged; control characters, spaces, most punctuation and 8-bit characters are converted into their hexadecimal value.
dataEncodeUTF7	This encoding method converts Unicode text into 7-bit characters that can be safely passed through systems that do not support Unicode or 8-bit characters. This is most commonly used with email applications.
dataEncodeUTF8	This encoding method converts Unicode text into 8-bit characters that can be safely passed through systems that do not support Unicode.

Data Type

Integer (Int32)

See Also

DecodedText Property, EncodedText Property, DecodeFile Method, EncodeFile Method

EncryptedText Property

Set the value of the current encrypted text string, or return the encrypted value of a plain text string.

Syntax

object.**EncryptedText** [= *value*]

Remarks

The **EncryptedText** property is used to specify the current encrypted text value, or return the decrypted value of a previously encrypted string. If the property is set, the current text is changed to this value and reading the **DecryptedText** property will return the decrypted string for this value. If you are assigning a value to this property, it must be a base64 encoded string.

The value of the **Password** property will be used to generate the encryption key. If the **Password** property has not been set, or if it's an empty string, an default internal hash value is used to encrypt the data. Password values that exceed 215 characters will be truncated.

Due to how the encryption key is created internally, another third-party library or component cannot be used to decrypt the string value returned by this property. The encryption is performed using the 256-bit AES (Advanced Encryption Standard) algorithm, and the key is generated using an SHA-256 hash of the password value. It is not possible to recover previously encrypted data if the password value is unknown.

Data Type

String

See Also

DecryptedText Property, Password Property, DecryptData Method, EncryptData Method, EncryptFile 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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

Password Property

Gets and sets the password used to encrypt and decrypt data.

Syntax

object.**Password** [= *value*]

Remarks

The **Password** property specifies the value which is used to generate the encryption and decryption key. If the **Password** property has not been set, or if it's an empty string, an default internal hash value is used to encrypt and decrypt the data. Password values that exceed 215 characters will be truncated.

Although it is not required for your application to use a password to encrypt the data, it is recommended. If no password is specified, any other application that uses this control will be able to decrypt the data. Passwords are case-sensitive and must match exactly, including the use of any spaces. If this property value is not identical to what was used to encrypt the data, attempts to decrypt the data will fail.

The encryption is performed using the 256-bit AES (Advanced Encryption Standard) algorithm, and the key is generated using an SHA-256 hash of the password value. It is not possible to recover previously encrypted data if the password value is unknown.

Data Type

String

See Also

DecryptedText Property, EncryptedText Property, DecryptFile Method, EncryptFile Method

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
FileEncoder1.ThrowError = False
nError = FileEncoder1.EncodeFile(strInputFile, strOutputFile)

If nError > 0 Then
    MsgBox FileEncoder1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

FileEncoder1.ThrowError = True
FileEncoder1.EncodeFile strInputFile, strOutputFile

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Version Property

Return the current version of the object.

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

File Encoding Control Methods

Method	Description
CompareFile	Compares the contents of two files with an optional message digest
CompressData	Compress the contents of a string or byte array
CompressFile	Compress the contents of the specified file
DecodeFile	Decode the contents of a file encoded by EncodeFile
DecryptData	Decrypt a string or byte array encrypted by EncryptData
DecryptFile	Decrypt the contents of a file encrypted by EncryptFile
EncodeFile	Encode the contents of the specified file
EncryptData	Encrypt the contents of a string or byte array
EncryptFile	Encrypt the contents of the specified file
ExpandData	Expand the contents of a string or byte array
ExpandFile	Expands the contents of a file compressed by CompressFile
Initialize	Initialize the control using the specified runtime license key
Reset	Reset the internal state of the control to its initial defaults
Uninitialize	Uninitialize the control

CompareFile Method

Compares the contents of two files with an optional message digest.

Syntax

object.**CompareFile**(*File1*, [*File2*], [*Options*], [*Digest*])

Parameters

File1

A string which specifies the name of the first file to compare. This parameter must specify the name of a file which exists and can be opened for read access by the current process. If the file name specifies a directory or device name, the method will fail. It is permitted to use environment variables in the file name by surrounding the variable name with the percent symbol. If the file name contains any leading or trailing space characters after the variable expansion, they will be removed. This parameter cannot be a zero-length string.

File2

An optional string which specifies the name of the second file to compare. If this parameter is omitted or a zero-length string, the ***Digest*** parameter must specify a valid message digest and the contents of the first file will be compared against that digest. If this parameter specifies a file name, the file must exist and the current process must be able to open it for read access. If the file name specifies a directory or device name, the method will fail. It is permitted to use environment variables in the file name by surrounding the variable name with the percent symbol. If the file name contains any leading or trailing space characters after the variable expansion, they will be removed.

Options

An integer value which specifies one or more option flags which will control how the files are compared. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
fileCompareDefault	Use the default comparison option, which is to perform a byte comparison of both files. If the ***Digest*** parameter is omitted or a zero-length string, the method will compare the message digest against the contents of the file. If the ***Digest*** parameter is included and no specific hash algorithm has been specified as one of the options, the hash type will be automatically determined based on its value.

fileCompareContents	Perform a byte comparison of the two files. This option requires both files be specified. The current process must be able to open both files for exclusive read access. If either file is opened for write access, the method will fail. The ***File2*** parameter must specify a valid file name if this option is used.	
fileCompareCreated	Compare the file creation times for both files and only consider them to be identical if the creation times are identical. If this option is specified and the creation times are different, the method will return an error even if the contents of the two files are identical. The ***File2*** parameter must specify a valid file name if this option is used.	
fileCompareModified	Compare the file modification times for both files and only consider them to be identical if the modification times are identical. If this option is specified and the modification times are different, the method will return an error even if the contents of the two files are identical. The ***File2*** parameter must specify a valid file name if this option is used.	
&H100	fileCompareMD5	Use the MD5 algorithm to compute 128-bit hash value for the contents of the first file and compare this against the specified digest string. This algorithm is the most efficient

		method to compute a hash to verify data integrity, however it is not considered cryptographically secure and hash collision is possible. When this option is used the ***Digest*** parameter must specify a valid MD5 hash as a 32 digit hexadecimal string.
&H200	fileCompareSHA1	Use the SHA-1 algorithm to compute 160-bit hash value for the contents of the first file and compare this against the specified digest string. This algorithm is comparable to the MD5 algorithm and has a lower chance of hash collision; however, it is not considered cryptographically secure and hash collision is possible. When this option is used the ***Digest*** parameter must specify a valid SHA-1 hash as a 40 digit hexadecimal string.
&H400	fileCompareSHA256	Use the SHA-256 algorithm to compute a 256-bit hash value for the contents of the first file and compare this against the specified digest string. This is a cryptographically secure algorithm which is more compute intensive than either MD5 or SHA-1, but the chance of a hash collision is extremely low and typically not a concern for most applications. When this option is used the ***Digest*** parameter must specify a valid SHA-256 hash as a 64 digit hexadecimal string.

Digest

An optional string which specifies a sequence of hexadecimal numbers which are used to compare against the computed hash value for the file. If this parameter is used and no option is specified to indicate the hash type, it will be inferred from the length of the string. If this parameter is omitted or a zero-length string, it will be ignored. The case of the hexadecimal digits in the string are ignored and it is permitted, but not required, to prefix the string with either "0x" or "&H" to indicate it is a hexadecimal value. If the string does not specify a valid hexadecimal number or is not a value computed using one of the supported hash algorithms, the method will fail.

Return Value

A return value of zero indicates there were no errors and the files are identical using the specified criteria. A non-zero return value specifies an error code which indicates the reason for the failure.

Remarks

The **CompareFile** method can be used to either compare the contents of two files or compare the contents of a file against a pre-computed hash value to determine if they match. In addition to comparing file contents, the method can also check the file creation and/or modification times to ensure they are the same.

In most cases, an application will specify either two files to compare or a message digest value. If you call the method using two file names and a message digest, the contents of both files must be identical and the computed hash value must match the specified message digest. If any of the requirements are are not satisfied, the method will return an error code. For example, if the actual contents of the two files are identical, but the value specified by the ***Digest*** parameter does not match the computed hash value of files, the method will still fail.

This method performs the file comparison in a way to minimize potential disk I/O whenever possible. First, the size of the two files are compared and if they are different, no further checks are performed. Next, if either the **fileCompareCreated** or **fileCompareModified** options have been specified, the file timestamps are compared and no further checks are performed if they do not match. A byte comparison of both files will only be performed after the other comparison criteria has been satisfied.

Comparing the contents of a file against a message digest value is the most expensive operation in terms of memory and processor utilization. The larger the files are, and the more complex the hash algorithm, the more compute-intensive the operation will be. In most cases, the SHA-1 algorithm provides a good balance between resource utilization and minimizing the possibility of hash collisions where different files could potentially generate the same hash value.

If it is critically important to minimize the possibility of a hash collision, use the SHA-256 algorithm; however, be aware of the impact it can have on performance when used with very large files. The current thread will block while the hash is being computed and if a large amount of data is being processed, this may cause the application to become non-responsive.

This method will normalize the file names provided by the caller and perform checks to make sure the names do not contain illegal characters. If the name includes quotes, wildcard characters or other invalid symbols the method will fail and return **stErrorInvalidFileName**. It is permitted to use forward slashes in path names and these will be converted to the standard backslash character used with Windows paths. This method cannot be used to compare the contents of NTFS alternate data streams.

Example

```
' Compare the contents of a file against an SHA-1 hash value
Dim strFileName As String
Dim strDigest String
Dim nError As Long

strFileName = "%USERPOFILE%\\Documents\\Projects\\MyDocument.docx"
strDigest = "541E43FDC612E41C02B770A2FD87EBB4A12EA800" ' SHA-1 hash value

nError = FileEncoder1.CompareFile(strFileName, , fileCompareSHA1, strDigest)

If nError = 0 Then
    MsgBox "The file contents match the SHA-1 digest", vbInformation
Else
    MsgBox "The file contents do not match the SHA-1 digest", vbExclamation
```

```
End If

' Compare the contents of two files using the default options
Dim strFile1 As String
Dim strFile2 As String
Dim nError As Long

strFile1 = "%USERPROFILE%\\AppName\\ProjectData.db"
strFile2 = "%TEMP%\\BackupData.db"

nError = FileEncoder1.CompareFile(strFile1, strFile2)

If nError = 0 Then
    MsgBox "The two files are identical", vbInformation
Else
    MsgBox "The two files are different", vbExclamation
End If
```

See Also

CompressFile Method, DecodeFile Method, EncodeFile Method, ExpandFile Method

CompressData Method

Compress the contents of the specified string or byte array.

Syntax

object.**CompressData**(*InputData*, *OutputData*)

Parameters

InputData

A string or byte array that contains the data to be compressed.

OutputData

A string or byte array that is passed by reference and will contain the compressed data when the method returns. If this parameter is a string or variant type, the compressed data will be automatically base64 encoded. If this parameter is a byte array, the compressed data will be copied into the array.

Return Value

This method returns the number of bytes of compressed data that was copied into the output buffer. If the input buffer is a zero-length string or an empty array, the method will return zero. If an error occurs, the method will return -1. Check the value of the **LastError** property to determine the specific error that has occurred.

Remarks

The **CompressData** method compresses the contents of a string or byte array in memory, rather than using a temporary file. The type of variable passed to the method as the output buffer determines whether the data is base644 encoded or not. If the output buffer is a string type then the compressed data will be automatically encoded to ensure that it can be safely represented as a string. If the output buffer is a byte array, the compressed data will be copied into the array as-is without any encoding. If the caller specifies a fixed-size byte array as the output buffer, the array must be large enough to contain all of the compressed data, otherwise the method will fail.

The return value from this method should always be checked to ensure that the data was successfully compressed. An application should never assume that the output buffer contains valid compressed data unless the return value is greater than zero. The compressed data is not returned in a format that is recognized by third-party applications such as PKZip or WinZip. To expand the compressed data, pass the contents of the output buffer to the **ExpandData** method.

See Also

CompressFile Method, DecryptData Method, EncryptData Method, ExpandData Method, ExpandFile Method

CompressFile Method

Compress the contents of the specified file.

Syntax

object.**CompressFile**(*InputFile*, *OutputFile*, [*CompressionType*], [*CompressionLevel*])

Parameters

InputFile

A string value that is the name of the file that will be compressed. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as the console, is specified as the file name.

OutputFile

A string value that is the name of the file that is to contain the compressed file data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as the console, is specified as the file name.

CompressionType

A numeric value which determines the algorithm that will be used to compress the data. One of the following values may be specified. If this argument is not specified, the Deflate algorithm is used.

Value	Description
fileCompressionDeflate	A compression algorithm that combines LZ77 algorithm for creating common substrings and Huffman coding to process the different frequencies of byte sequences in the data stream. Deflate is widely used by compression software and provides a good balance between the data compression ratio and system resources used. This is the default compression method.
fileCompressionBzip2	A compression algorithm that rearranges blocks of data in sorted order and then uses Huffman coding to process different frequencies of data within the block. Burrows-Wheeler compression provides a better compression ratio than the Deflate algorithm, however it requires more resources to perform the compression.
fileCompressionLzma	A compression algorithm that combines the LZ77 algorithm for dictionary-based compression with range encoding to efficiently represent repetitive patterns in the data. LZMA can achieve higher compression ratios than Deflate or the Burrows-Wheeler algorithms; however, it requires more memory and CPU resources due to the larger dictionary sizes and more complex encoding algorithm.

CompressionLevel

A numeric value which specifies the compression level to use. A value of zero specifies that the default compression level appropriate for the selected algorithm should be used, balancing resource usage and the compression ratio of the data. A value of 1 specifies that the compression

should be performed using minimal memory resources, at the expense of the compression ratio. The maximum value of 9 specifies that the algorithm should use more memory to achieve the maximum compression ratio. It is recommended that most applications use the default value of zero.

Return Value

This method returns a value of zero if the file was successfully compressed. A non-zero return value specifies an error code which indicates the reason for the failure.

Remarks

The **CompressFile** method compresses the contents of the specified file. The compression ratio achieved depends on the type of file being compressed. Note that the compressed file is not stored in an archive format that is recognized by third-party applications such as PKZip or WinZip.

If the output file already exists, it will be replaced with the compressed contents of the input file. If the compression fails for any reason, the output file will be deleted. If you want to ensure an existing output file will be preserved if an error occurs, you should create a temporary file and use it as the output file. If this method succeeds, then you can rename or copy the temporary file which contains the compressed data.

See Also

CompareFile Method, CompressData Method, DecodeFile Method, EncodeFile Method, ExpandFile Method

DecodeFile Method

Decode the contents of the specified file.

Syntax

object.**DecodeFile**(*InputFile*, *OutputFile*, *Encoding*)

Parameters

InputFile

A string value that specifies the name of the file to be decoded. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as the console, is specified as the file name.

OutputFile

A string value that specifies the name of the file that will contain the decoded data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as the console, is specified as the file name.

Encoding

The encoding method that was used to create the file. The following encoding methods are valid:

Value	Description
fileDecodeDefault	Use the default encoding method. Currently this is the same specifying that the base64 algorithm should be used for encoding and decoding files.
fileDecodeBase64	Use the base64 algorithm for encoding and decoding files. This is the standard method for encoding files as outlined in the Multipurpose Internet Mail Extensions (MIME) protocol. This is the method used by most modern email client software.
fileDecodeQuoted	This encoding method is typically used for text messages that use characters beyond the standard ASCII character set, in the range of 128-255. This method, called quoted printable encoding, allows text messages to pass through mail systems that do not support characters with the high-bit set. Note that this method should not be used to encode binary files such as executables or file archives.
fileDecodeUucode	Use the uuencode and uudecode algorithms for encoding and decoding files. This is a common encoding method used with UNIX systems and older email client software.
fileDecodeYencode	Use the yEnc algorithm for encoding the file. This is an encoding method that is commonly used when posting files to Usenet newsgroups.

&H10000	fileDecodeCompressed	This option is used in combination with one of the encoding types listed above. If specified, the method will decode the file and expand the data, restoring the original file contents.

Return Value

This method returns a value of zero if the file was successfully decoded. A non-zero return value specifies an error code which indicates the reason for the failure.

Remarks

The **DecodeFile** method decodes the contents of a file that was created using the specified encoding method.

The **fileDecodeCompressed** option should only be specified if the encoded file was created using **EncodeFile** method with the **fileEncodeCompressed** option.

See Also

Encoding Property, CompareFile Method, CompressFile Method, EncodeFile Method, ExpandFile Method

DecryptData Method

Decrypt the contents of the specified string or byte array.

Syntax

object.**DecryptData**(*InputData*, *OutputData* [, *Password*])

Parameters

InputData

A string or byte array that contains the data to be decrypted.

OutputData

A string or byte array which will contain the decrypted data when the method returns. This parameter must be passed by reference. When specifying a **Byte** array, you must ensure the buffer is large enough to contain all of the decrypted data.

Password

An optional parameter that specifies the password that was used to decrypt the data. If this parameter is omitted, the value of the **Password** property will be used.

Return Value

This method returns the number of bytes of decrypted data copied into the output buffer. If the input buffer is a zero-length string or an empty array, the method will return zero. If an error occurs, the method will return -1. Check the value of the **LastError** property to determine the specific error that has occurred.

Remarks

The **DecryptData** method will decrypt the contents of the input buffer previously encrypted with the **EncryptData** method. Thee decrypted data is returned in the specified output buffer. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the decryption process.

The input and output buffer variables must match the same data types which were used when calling the **EncryptData** method. If it was used to encrypt a string, then the input and output variables must be **String** types. If it was used to encrypt binary data, the input and output variables must be **Byte** arrays. Never attempt to encrypt or decrypt binary data using **String** variables. You must always use a **Byte** array for binary data.

Due to how the SHA-256 hash is generated, this method cannot be used to decrypt files that were encrypted using another third-party library. It can only be used to decrypt data that was previously encrypted using **EncryptData**.

If you wish to encrypt and decrypt the contents of a file, use the **EncryptFile** and **DecryptFile** methods.

This method uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries. If the required cryptographic provider is not available, the method will fail.

See Also

CompressFile Method, EncryptData Method, ExpandData Method, ExpandFile Method

DecryptFile Method

Decrypt the contents of the specified file.

Syntax

object.**DecryptFile**(*InputFile*, *OutputFile* [, *Password*])

Parameters

InputFile

A string value that specifies the name of the file to be decrypted. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as the console, is specified as the file name.

OutputFile

A string value that specifies the name of the file that will contain the decrypted data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as the console, is specified as the file name.

Password

An optional parameter that specifies the password that was used to encrypt the file contents. If this parameter is omitted, the value of the **Password** property will be used.

Return Value

This method returns a value of zero if the file was successfully decrypted. A non-zero return value specifies an error code which indicates the reason for the failure.

Remarks

The **DecryptFile** method will decrypt the contents of a file previously encrypted with the **EncryptFile** method and stores the decrypted data in the specified output file. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the decryption process.

Due to how the SHA-256 hash is generated, this method cannot be used to decrypt files that were encrypted using another third-party library. It can only be used to decrypt data that was previously encrypted using **EncryptFile**.

A temporary file is created during the decryption process and the output file is created or overwritten only if the input file could be successfully decrypted. If the decryption fails, no output file will be created.

This method uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries. If the required cryptographic provider is not available, the method will fail.

See Also

DecryptedText Property, EncryptedText Property, Password Property, EncryptData Method, EncryptFile Method

EncodeFile Method

Encode the contents of the specified file.

Syntax

object.**EncodeFile**(*InputFile*, *OutputFile*, *Encoding*)

Parameters

InputFile

A string value that specifies the name of the file to be encoded. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as CON: is specified as the file name.

OutputFile

A string value that specifies the name of the file that will contain the encoded data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as CON: is specified as the file name.

Encoding

The encoding method to be used when creating the file. The following encoding methods are valid:

Value	Description
fileEncodeDefault	Use the default encoding method. Currently this is the same specifying that the base64 algorithm should be used for encoding and decoding files.
fileEncodeBase64	Use the base64 algorithm for encoding and decoding files. This is the standard method for encoding files as outlined in the Multipurpose Internet Mail Extensions (MIME) protocol. This is the method used by most modern email client software.
fileEncodeQuoted	This encoding method is typically used for text messages that use characters beyond the standard ASCII character set, in the range of 128-255. This method, called quoted printable encoding, allows text messages to pass through mail systems that do not support characters with the high-bit set. Note that this method should not be used to encode binary files such as executables or file archives.
fileEncodeUucode	Use the uuencode and uudecode algorithms for encoding and decoding files. This is a common encoding method used with UNIX systems and older email client software.
fileEncodeYencode	Use the yEnc algorithm for encoding the

	file. This is an encoding method that is commonly used when posting files to Usenet newsgroups.	
&H10000	fileEncodeCompressed	This option is used in combination with one of the encoding types listed above. If specified, the file will be compressed prior to being encoded by the control. This can significantly reduce the size of the encoded output.

Return Value

This method returns a value of zero if the file was successfully encoded. A non-zero return value specifies an error code which indicates the reason for the failure.

Remarks

The **EncodeFile** method encodes the contents of a file, using the specified encoding method.

When specifying the **fileEncodeCompressed** option, it is important to remember that the compressed, encoded data can only be restored to its original contents using the **DecodeFile** method. This option should not be used when encoding a file to be attached to an email message unless you provide the recipient with a utility to decode the data.

See Also

Encoding Property, CompareFile Method, CompressFile Method, DecodeFile Method, ExpandFile Method

EncryptData Method

Encrypt the contents of the specified string or byte array.

Syntax

object.**EncryptData**(*InputData*, *OutputData* [, *Password*])

Parameters

InputData

A string or byte array that contains the data to be encrypted.

OutputData

A string or byte array which will contain the encrypted data when the method returns. This parameter must be passed by reference. When specifying a **Byte** array, you must ensure the buffer is large enough to contain all of the decrypted data.

Password

An optional parameter that specifies the password that was used to encrypt the data. If this parameter is omitted, the value of the **Password** property will be used.

Return Value

This method returns the number of bytes of encrypted data copied into the output buffer. If the input buffer is a zero-length string or an empty array, the method will return zero. If an error occurs, the method will return -1. Check the value of the **LastError** property to determine the specific error that has occurred.

Remarks

The **EncryptData** method encrypts the contents of a string or byte array using a 256-bit AES (Advanced Encryption Standard) algorithm and stores the encrypted data in the specified output file. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the encryption process. The identical password is required to decrypt the data using the **DecryptData** method.

Although it is not required for your application to use a password to encrypt the data, it is recommended. If no password is specified, any other application that uses this control will be able to decrypt the data. Passwords are case-sensitive and must match exactly, including the use of any spaces.

If the input buffer is a **String** type, the output buffer must also be a **String**. The text will be automatically encoded as UTF-8 and the encrypted data will be returned as a base64 encoded string. If the input buffer is a **Byte** array, the output buffer must also be a **Byte** array. Never attempt to encrypt or decrypt binary data using **String** variables. You must always use a **Byte** array for binary data.

If you wish to encrypt and decrypt the contents of a file, use the **EncryptFile** and **DecryptFile** methods.

This method uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries. If the required cryptographic provider is not available, the method will fail.

See Also

CompressFile Method, DecryptData Method, ExpandData Method, ExpandFile Method

EncryptFile Method

Encrypt the contents of the specified file.

Syntax

object.**EncryptFile**(*InputFile*, *OutputFile* [, *Password*])

Parameters

InputFile

A string value that specifies the name of the file to be encrypted. The file must exist, and it must be a regular file that can be opened for reading by the current process. An error will be returned if a character device, such as the console, is specified as the file name.

OutputFile

A string value that specifies the name of the file that will contain the encrypted data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as the console, is specified as the file name.

Password

An optional parameter that specifies the password that was used to encrypt the file contents. If this parameter is omitted, the value of the **Password** property will be used.

Return Value

This method returns a value of zero if the file was successfully encrypted. A non-zero return value specifies an error code which indicates the reason for the failure.

Remarks

The **EncryptFile** method will encrypt the contents of a file using a 256-bit AES (Advanced Encryption Standard) algorithm and stores the encrypted data in the specified output file. The password (or passphrase) provided by the caller is used to generate a SHA-256 hash value which is used as part of the encryption process. The identical password is required to decrypt the data using the **DecryptFile** method.

Although it is not required for your application to use a password to encrypt the data, it is recommended. If no password is specified, any other application that uses this control will be able to decrypt the data. Passwords are case-sensitive and must match exactly, including the use of any spaces.

A temporary file is created during the encryption process and the output file is created or overwritten only if the input file could be successfully encrypted. If the encryption fails, no output file will be created.

If you wish to encrypt or decrypt string values, use the **DecryptedText** and **EncryptedText** properties.

This method uses the Microsoft CryptoAPI and the RSA AES cryptographic provider. This provider may not be available in some languages, countries or regions. The availability of this provider may also be constrained by cryptography export restrictions imposed by the United States or other countries. If the required cryptographic provider is not available, the method will fail.

See Also

DecryptedText Property, EncryptedText Property, Password Property, DecryptData, EncryptData, EncryptFile Method

---

ExpandData Method

Expand the contents of the specified string or byte array.

Syntax

object.**ExpandData**(*InputData*, *OutputData*)

Parameters

InputData

A string or byte array that contains the data to be expanded.

OutputData

A string or byte array that is passed by reference and will contain the expanded data when the method returns.

Return Value

This method returns the number of bytes of expanded data that was copied into the output buffer. If the input buffer is a zero-length string or an empty array, the method will return zero. If an error occurs, the method will return -1. Check the value of the **LastError** property to determine the specific error that has occurred.

Remarks

The **ExpandData** method expands data that compressed with a previous call to the **CompressData** method. If the input buffer is a string of base64 encoded data, it will automatically be decoded prior to being expanded. If the expanded (uncompressed) data is textual, then the output buffer can be a string or variant type. If the expanded data is binary, the output buffer should always be a byte array. Binary data that is expanded into a string buffer may be corrupted because the data will be automatically be converted to Unicode by this method. This conversion is not performed if the output buffer is a byte array. If the caller specifies a fixed-size byte array as the output buffer, the array must be large enough to contain all of the expanded data, otherwise the method will fail.

The return value from this method should always be checked to ensure that the data was successfully expanded. An application should never assume that the output buffer contains valid data unless the return value is greater than zero.

See Also

CompressData Method, CompressFile Method, DecryptData Method, EncryptData Method, ExpandFile Method

ExpandFile Method

Expands the contents of a previously specified file.

Syntax

object.**ExpandFile**(*InputFile*, *OutputFile*, [*CompressionType*])

Parameters

InputFile

A string value that specifies the name of the file to be expanded. The file must exist, and it must be a file that was created by a previous call to the **CompressFile** method. An error will be returned if the file cannot be accessed, if it specifies a character device such as the console, or the if the data is not in a recognizable format.

OutputFile

A string value that specifies the name of the file that will contain the expanded data. If the file exists, it must be a regular file that can be opened for writing by the current process and will be overwritten. If the file does not exist, it will be created. An error will be returned if a character device, such as the console, is specified as the file name.

CompressionType

A numeric value which determines the algorithm that was used to compress the data. One of the following values may be specified. If this argument is not specified, the Deflate algorithm is used. If the compression type specified by this argument does not match the actual compression algorithm used to compress the file, an error will be returned.

Value	Description
fileCompressionUnknown	The method should attempt to determine what compression algorithm was used when the data was compressed. This is done by examining the header block of the file and checking for certain signatures which can identify the algorithm. If the algorithm cannot be determined, the method will fail. It is recommended that most applications explicitly specify the compression algorithm.
fileCompressionDeflate	A compression algorithm that combines LZ77 algorithm for creating common substrings and Huffman coding to process the different frequencies of byte sequences in the data stream. Deflate is widely used by compression software and provides a good balance between the data compression ratio and system resources used. This is the default compression method.
fileCompressionBzip2	A compression algorithm that rearranges blocks of data in sorted order and then uses Huffman coding to process different frequencies of data within the block. Burrows-Wheeler compression provides a better compression ratio than the Deflate algorithm, however it requires more resources to perform the

	compression.
fileCompressionLzma	A compression algorithm that combines the LZ77 algorithm for dictionary-based compression with range encoding to efficiently represent repetitive patterns in the data. LZMA can achieve higher compression ratios than Deflate or the Burrows-Wheeler algorithms; however, it requires more memory and CPU resources due to the larger dictionary sizes and more complex encoding algorithm.

Return Value

This method returns a value of zero if the file was successfully decompressed. A non-zero return value specifies an error code which indicates the reason for the failure.

Remarks

The **ExpandFile** method expands the contents of a previously compressed file. Note that this method can only expand files that were compressed using the control. It cannot expand the contents of a file stored in an archive format such as PKZip or WinZip.

The value of the ***CompressionType*** parameter must match the value which was used with the **CompressFile** method. This method will fail if you specify a different compression algorithm than what was used to compress the file. Although your application can use **fileCompressionUnknown** and attempt to automatically determine how the file was compressed, this is not always reliable.

If the output file already exists, it will be replaced with the expanded contents of the input file. If the contents of the file cannot be expanded for any reason, the output file will be deleted. If you want to ensure an existing output file will be preserved if an error occurs, you should create a temporary file and use it as the output file. If this method succeeds, then you can rename or copy the temporary file which contains the expanded data.

See Also

CompareFile Method, CompressFile Method, DecodeFile Method, EncodeFile 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 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 fileEncoder = CreateObject("SocketTools.FileEncoder.11")

nError = fileEncoder.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

Reset Method, Uninitialize 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 and any handles allocated by the control will be released.

See Also

Initialize Method, Uninitialize 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 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

File Encoding Control Events

Event	Description
OnError	This event is generated when a control error occurs
OnProgress	This event is generated the file contents are encoded or decoded

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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger this event.

The ***ErrorCode*** argument specifies the last error that has occurred.

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 the file contents are encoded or decoded.

Syntax

Sub *object*_**OnProgress** ([*Index* **As Integer**,] **ByVal** *FileName* **As Variant**, **ByVal** *FileSize* **As Variant**, **ByVal** *FileProcessed* **As Variant**, **ByVal** *Percent* **As Variant**)

Remarks

The **OnProgress** event is generated as files are being encoded or decoded. For large files, 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:

FileName

The name of the file currently being encoded or decoded.

FileSize

The size of the file in bytes.

FileProcessed

The number of bytes in the file which have been encoded or decoded.

Percent

The percentage of data in the file which has been encoded or decoded, expressed as an integer value between 0 and 100, inclusive.

Note that this event is guaranteed to fire at least twice, immediately before the encoding or decoding process begins, and after it has completed. To prevent an application from being flooded with **OnProgress** events during the course of the encoding process, the control meters the time intervals at which this event is triggered. Unless the file is very large, it is not uncommon for interim events to not fire during the decoding or encoding process.

See Also

DecodeFile Method, EncodeFile Method

File Transfer Control

Transfer files between a local and server and perform common file management functions on the server.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	FileTransferCtl.FileTransfer
File Name	CSFTCX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.FileTransfer.11
ClassID	113C9358-86CA-4A6A-8601-8A367F99865E
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 959, RFC 1579, RFC 1945, RFC 2228, RFC 2616

Overview

The SocketTools File Transfer ActiveX control provides a simplified interface for uploading and downloading files over the Internet or an intranet. The control implements the File Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP) for transferring files between the local system and a server.

An application can transfer a file with a single method call, simply by specifying a URL, without the need to provide the individual protocol, host, port, file name, and account information. Alternatively, connection and access information may be supplied separately, to allow multiple file transfer operations to be performed in a single server session. In either case, the differences between the supported protocols are kept to a minimum. Additional features such as proxy connections are easily implemented by simply setting a few properties.

The control offers a comprehensive interface that provides everything needed to incorporate file transfer functionality in an application, as well as perform remote file management. In addition to downloading and uploading by file name, URL, and wild card patterns, a developer may use the control for the creation, listing, and removal of directories, as well as renaming and removal of files on the server.

In addition to supporting standard FTP and HTTP connections, the File Transfer Control also supports secure TLS 1.2 and SFTP (SSH 2.0) connections. By simply setting a few properties, a secure connection using up to 256-bit AES encryption can be established, providing your application with the greatest flexibility and highest level of security available.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

File Transfer Control Properties

Property	Description
Account	Gets and sets the FTP account name for the current user
ActivePort	Gets and sets the range of ports used for active mode file transfers
AppendFile	Specify that data should be appended to an existing file during a FTP file transfer
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 client certificate
CertificatePassword	Gets and sets the password associated with the client certificate
CertificateStatus	Return the status of the server certificate
CertificateStore	Gets and sets the name of the client 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
ChannelMode	Gets and sets the security mode for the specified communications channel
CipherStrength	Return the length of the key used by the encryption algorithm
CodePage	Gets and sets the code page used for Unicode text conversion
Compression	Set or return if data compression should be enabled for HTTP downloads
DirectoryFormat	Gets and sets the current FTP directory format type
Features	Gets and sets the features enabled for the current client session
FileType	Gets and sets the current file transfer type for FTP transfers
Fingerprint	Returns a string that uniquely identifies the server
HashStrength	Return the length of the message digest that was selected
IsBlocked	Determine if the control is blocked performing an operation
IsConnected	Determine if the control is connected to a server
IsInitialized	Determine if the control has been initialized
KeepAlive	Set or return if the connection to a HTTP server is persistent
LastError	Gets and sets the last error code
LastErrorString	Return a description of the last error that occurred
Localize	Determines if remote file dates are localized to the current timezone
Passive	Enable passive FTP file transfers
Password	Gets and sets the password for the current user
Priority	Gets and sets the priority assigned to file transfers
PrivateKey	Gets and sets the private key file used for SSH authentication
ProtocolVersion	Gets and sets the current HTTP protocol version

ProxyPassword	Gets and sets the proxy server password for the current user
ProxyPort	Gets and sets the port number for the proxy server
ProxyServer	Gets and sets the host name of the proxy server
ProxyType	Gets and sets the current proxy server type
ProxyUser	Gets and sets the current proxy user name
ResultCode	Return the result code of the previous action
ResultString	Return a string describing the results of the previous action
Resource	Gets and sets the remote file name or resource path on the server
Secure	Specify if a connection to the server is secure
SecureCipher	Return the encryption algorithm used to establish the secure connection with the server
SecureHash	Return the message digest selected when establishing the secure connection with the server
SecureKeyExchange	Return the key exchange algorithm used to establish the secure connection with the server
SecureProtocol	Gets and sets the security protocol used to establish the secure connection with the server
ServerDirectory	Gets and sets the current working directory on the FTP server
ServerName	Gets and sets the host name for the FTP or HTTP server
ServerPort	Gets and sets the port number for a remote connection
ServerType	Gets and sets the type of the remote FTP or HTTP server
System	Return information about the server
TaskCount	Return the number of active background file transfers
TaskId	Return the task ID for an active background file transfer
TaskList	Return the task ID for an active background file transfer
ThrowError	Enable or disable error handling by the container of the control
Timeout	Gets and sets the amount of time until a blocking network operation is aborted
Trace	Enable or disable network function level tracing
TraceFile	Return or specify the network function trace output file
TraceFlags	Gets and sets the current network function tracing flags
TransferBytes	Return the number of bytes transferred from the server
TransferBytesXL	Return the number of bytes transferred from the server
TransferRate	Return the current data transfer rate in bytes per second
TransferTime	Return the number of seconds elapsed during a data transfer
URL	Gets and sets the current Uniform Resource Locator value
UserName	Gets and sets the current user name
Version	Return the current version of the object

Account Property

Gets and sets the FTP account name for the current user.

Syntax

object.**Account** [= *account*]

Remarks

The **Account** property specifies the account name of the current user, if it is required by the server for authentication.

Note that not all servers require an account name, in which case this property is ignored.

Data Type

String

See Also

Password Property, UserName Property, Connect Method

ActivePort Property

Gets and sets the range of local port numbers used for active mode FTP file transfers.

Syntax

object.**ActivePort**(*portrange*) [= *localport*]

Remarks

The **ActivePort** property property array is used to change the range of local port numbers used for active mode file transfers. The property array index specifies the port that should be changed, and may be one of the following values:

Value	Description
ftpActivePortLow	Change or return information for the low port number.
ftpActivePortHigh	Change or return information for the high port number.

The ***localport*** value specifies the new port number to be used. Valid port numbers are in the range of 1025 through 65535.

This property array is used to modify the range of local port numbers used for active mode file transfers. When using active mode, the client listens for an inbound connection from the server rather than establishing an outbound connection for the data transfer. In most cases, passive mode transfers are preferred because they mitigate potential compatibility issues with firewalls and NAT routers.

If active mode transfers are required, the default port range used when listening for the server connection is between 1024 and 5000. This is the standard range of ephemeral ports used by the Windows operating system. However, under some circumstances that range of ports may be too small, or a firewall may be configured to deny inbound connections on ephemeral ports. In that case, the **ActivePort** property can be used to specify a different range of port numbers.

While it is technically permissible to assign the low and high port numbers to the same value, effectively specifying a single active port number, this is not recommended as it can cause the transfer to fail unexpectedly if multiple file transfers are performed. A minimum range of at least 1000 ports is recommended. For example, if you specify a low port value of 40000 then it is recommended that the high port value be at least 41000. The maximum port value is 65535.

Data Type

Integer (Int32)

See Also

Features Property, Connect Method, Disconnect Method

AppendFile Property

Specify that data should be appended to an existing file during a FTP file transfer.

Syntax

object.**AppendFile** [= { True | False }]

Remarks

The **AppendFile** property specifies that data should be appended to the target file during a file transfer. The default value for this property is False, which means that the target file will be overwritten. Note that this property only has an effect when the **GetFile** or **PutFile** methods are used for a FTP transfer.

Data Type

Boolean

See Also

GetFile Method, PutFile Method

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

```
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

Dim strIssuer As String
Dim strCompanyName As String

strIssuer = FileTransfer1.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 client certificate.

Syntax

object.**CertificateName** [= *name*]

Remarks

This property sets the common name or friendly name of the certificate that should be used to establish the connection with the server. 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. If a certificate name is specified, the certificate must have a private key associated with it, otherwise the connection attempt 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 client 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
stCertificateNone	No certificate information is available. A secure connection was not established with the server.
stCertificateValid	The certificate is valid.
stCertificateNoMatch	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 **ServerName** property is set to an IP address rather than a host name. It is recommended that the client examine the **CertificateSubject** property to determine the domain name of the site that the certificate was issued for.
stCertificateExpired	The certificate has expired and is no longer valid. The client can examine the **CertificateExpires** property to determine when the certificate expired.
stCertificateRevoked	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.
stCertificateUntrusted	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.
stCertificateInvalid	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 and retrieves a file:

```
FileTransfer1.HostName = strHostName

On Error Resume Next: Err.Clear
FileTransfer1.Secure = True

If Err.Number Then
```

```
    MsgBox "Unable to initialize the security interface"
    Exit Sub
End If

On Error GoTo 0

nError = FileTransfer1.Connect()

If nError > 0 Then
    MsgBox "Unable to connect to server " & strHostName, vbExclamation
    Exit Sub
End If

If FileTransfer1.CertificateStatus <> stCertificateValid Then
    lResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                     "Are you sure you wish to continue?", vbYesNo)

    If lResult = vbNo Then
        FileTransfer1.Disconnect
        Exit Sub
    End If
End If

nError = FileTransfer1.GetFile(strLocalFile, strRemoteFile)
If nError > 0 Then
    FileTransfer1.Disconnect
    MsgBox "Unable to retrieve file from server " & strHostName
    Exit Sub
End If

FileTransfer1.Disconnect
```

See Also

CertificateExpires Property, CertificateIssued Property, CertificateIssuer Property, CertificateSubject Property, Secure Property

CertificateStore Property

Gets and sets the name of the client certificate store or file.

Syntax

object.**CertificateStore** [= *store*]

Remarks

This property sets the name of the certificate store that contains the client certificate that should be used when establishing a secure connection with 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 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

CertificateSubject Property

Returns information about the organization to which the server certificate was issued.

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 = FileTransfer1.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 client certificate.

Syntax

object.**CertificateUser** [= *username*]

Remarks

This property sets the name of the user that owns the client certificate that will be used to establish a secure connection with the server. 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

ChannelMode Property

Gets and sets the security mode for the specified communications channel.

Syntax

object.**ChannelMode**(*channel*) [= *mode*]

Remarks

The **ChannelMode** property property array is used to change the security mode for either the command or data channel. The property array index specifies the channel that should be changed, and may be one of the following values:

Value	Description
ftpChannelCommand	Change or return information for the command channel. This is the communication channel used to send commands to the server and receive command result and status information from the server.
ftpChannelData	Change information for the data channel. This is the communication channel used to send or receive data during a file transfer.

The *mode* value specifies the new security mode for the specified channel. It may be one of the following values:

Value	Description
ftpChannelClear	Data sent and received on this channel should not be encrypted.
ftpChannelSecure	Data sent and received on this channel should be encrypted. Specifying this option requires that a secure connection has already been established with the server.

The **ChannelMode** property array is used to change the default mode for the specified channel, and is typically used to control whether or not data is encrypted during a file transfer. If a standard, non-secure connection has been established with the server, an error will be returned if you specify the **ftpChannelSecure** mode for either channel.

If you have established a secure connection and then specify the **ftpChannelClear** mode for the command channel, the client will send the CCC command to the server to indicate that commands should no longer be encrypted. If the server does not support this command, an error will be returned and the channel mode will remain unchanged. Once the command channel has been changed to clear mode, it cannot be changed back to secure mode. You must disconnect and re-connect to the server if you want to resume sending commands over an encrypted channel.

Changing the mode for the data channel requires that the server support the PROT command. If this command is not supported by the server, an exception will be thrown which must be handled by the application. You can only set a channel to secure mode if the **Secure** property is also set to True.

It is important to note that this property array should only be used after a connection has been established with the server. If you attempt to read the property or change a value prior to calling the **Connect** method, an exception will be thrown.

Data Type

Integer (Int32)

See Also

Features Property, Secure Property, Connect Method, Disconnect Method

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 converting text to and from Unicode.

Syntax

object.**CodePage** [= *value*]

Remarks

The **CodePage** property is an integer value which specifies how text is encoded. Any valid code page identifier may be specified. Some common values are:

Value	Description
Text sent and received using a string should be converted using the ANSI code page for the current locale.	
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.	
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.	
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.	
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.	
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 exchanged with an FTP server 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, with each character represented using 16 bits. When you send and receive data using the String data type, they will automatically be converted to a stream of bytes.

By default, strings are converted to an array of bytes using UTF-8 encoding, mapping the 16-bit

Unicode characters to 8-bit bytes. Similarly, when reading data 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 the text you receive appears to corrupted or characters are being replaced with question marks or other symbols, it is likely the file on the server is using a different character encoding. Most applications use UTF-8 encoding to represent non-ASCII characters; however, some text files may use a localized character set rather than using Unicode. Using the **GetText** and **PutText** methods in combination with this property will change how that text is converted to Unicode.

Strings are only guaranteed to be safe when sending and receiving text. Using a string data type is not recommended when uploading or downloading binary data. If possible, you should always use a byte array when using the **GetData** and **PutData** methods.

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.

Although strings in Visual Basic are internally managed as Unicode, the default common controls used in Visual Basic 6.0 do not support Unicode. Those controls, such as buttons, text boxes and labels, will automatically convert the Unicode text to ANSI using the current code page. This means that text in the end-user's native language (depending on system settings) may display correctly, although text in other languages using different character sets may not. Also note that the VB6 IDE is not Unicode aware and may display corrupted string values or invalid characters, such as with tooltip values when debugging.

For Unicode support in Visual Basic 6.0, it's recommended that you use third-party controls. An alternative that some developers have used is the Microsoft Forms 2.0 Object Library (FM20.DLL) that is part of Microsoft Office. It includes a collection of controls that support Unicode, however they are not redistributable and Microsoft has stated that their use with VB6 is unsupported.

Data Type

Integer (Int32)

See Also

FileType Property, GetData Method, GetText Method, PutData Method, PutText Method

Compression Property

Set or return if data compression should be enabled.

Syntax

object.**Compression** [= { True | False }]

Remarks

The **Compression** property is used to indicate to the server whether or not it is acceptable to compress the data that is returned to the client. If compression is enabled, the client will advertise that it will accept compressed data and the server will decide whether a resource being requested can be compressed. If the data is compressed, the control will automatically expand the data before returning it to the caller.

Enabling compression does not guarantee that the data returned by the server will actually be compressed, it only informs the server that the client is willing to accept compressed data. Whether or not a particular resource is compressed depends on the server configuration, and the server may decide to only compress certain types of resources, such as text files. Disabling compression informs the server that the client is not willing to accept compressed data; this is the default.

This property value is only meaningful when downloading files from an HTTP server that supports file compression. It has no effect on file uploads or file transfers using FTP.

Data Type

Boolean

See Also

GetData Method, GetFile Method

DirectoryFormat Property

Gets and sets the current FTP directory format type.

Syntax

object.**DirectoryFormat** [= *format*]

Remarks

Value	Description
ftpDirectoryAutoDetect	This value specifies that the library should automatically determine the format of the file lists returned by the server. It is recommended that most applications use this value and allow the control to automatically determine the appropriate file listing format used by the server. This is the default value of the property.
ftpDirectoryUnix	This value specifies that the server returns file lists in the format commonly used by UNIX servers. Note that many servers can be configured to return file listings in this format, even if they are not actually a UNIX based platform. Consult the technical reference documentation for your server for more information.
ftpDirectoryMsdos	This value specifies that the server returns file lists in the format commonly used by MS-DOS based systems. This includes Windows IIS servers. Long file names will be returned if supported by the underlying filesystem, such as NTFS or FAT32.
ftpDirectoryVms	This value specifies that the server returns file lists in the format commonly used by VMS servers. Note that VMS servers can be configured to return a standard UNIX style listing in additional to the default VMS format.
ftpDirectorySterling1	This value specifies that the server returns file listings in a proprietary format used by the Sterling server, which is used for EDI (Electronic Data Interchange) applications. This format uses a 13 byte status code.
ftpDirectorySterling2	This value specifies that the server returns file listings in a proprietary format used by the Sterling server, which is used for EDI (Electronic Data Interchange) applications. This format uses a 10 byte status code.
ftpDirectoryNetWare	This value specifies that the server returns file listings in a proprietary format used by NetWare servers. The format is similar to UNIX style listings except that file access and permissions are indicated by letter codes enclosed in brackets. This is the default format selected if the server identifies itself as a NetWare system.

If this property has the default value **ftpDirectoryAutoDetect** initially, and the control can determine from the format of the first file in the listing that one of the other supported types is used, then the property will change value upon the first call to the **ReadDirectory** method. If the control cannot determine the format of the directory listing, then the directory listing will be empty. In this case, you can retrieve a directory listing by setting the optional ***ParseList*** parameter of the **OpenDirectory** method to False. Each line of the directory listing provided by the server will be returned in ***FileName*** parameter of the **ReadDirectory** function. It will then be the responsibility of the application to extract desired information from this string.

Data Type

Integer (Int32)

See Also

OpenDirectory Method, ReadDirectory Method

Features Property

Gets and sets the FTP features enabled for the current client session.

Syntax

object.**Features** [= *flags*]

Remarks

The **Features** property returns a value which may be a combination of one or more of the following bit flags:

Value	Description	
&H00001	ftpFeatureSIZE	The server supports the SIZE command to determine the size of a file. If this feature is not enabled, the control will attempt to use the MLST or STAT command to determine the file size.
&H00002	ftpFeatureSTAT	The server supports using the STAT command to return information about a specific file. If this feature is not enabled, the client may not be able to obtain information about a specific file such as its size, permissions or modification time.
&H00004	ftpFeatureMDTM	The server supports the MDTM command to obtain information about the modification time for a specific file. This command may also be used to set the file time on the server.
&H00008	ftpFeatureREST	The server supports restarting file transfers using the REST command. If this feature is not enabled, the client will not be able to restart file transfers and must upload or download the complete file.
&H00010	ftpFeatureSITE	The server supports site specific commands using the SITE command. If this feature is not enabled, no site specific commands will be sent to the server.
&H00020	ftpFeatureIDLE	The server supports setting the idle timeout period using the SITE IDLE command to specify the number of seconds that the client may idle before the server terminates the connection.
&H00040	ftpFeatureCHMOD	The server supports modifying the permissions of a specific file using the SITE CHMOD command. If this feature is not enabled, the client will not be able to set the permissions for a file.
&H00080	ftpFeatureAUTH	The server supports explicit TLS sessions using the AUTH command. If this feature is not enabled, the client will only be able to connect to a secure server that uses implicit TLS connections. Changing this feature has no effect on standard, non-secure connections.
&H00100	ftpFeaturePBSZ	The server supports the PBSZ command which specifies

		the buffer size used with secure data connections. If this feature is disabled, it may prevent the client from changing the protection level on the data channel. Changing this feature has no effect on standard, non-secure connections.
&H00200	ftpFeaturePROT	The server supports the PROT command which specifies the protection level for the data channel. If this feature is disabled, the client will be unable to change the protection level on the data channel. Changing this feature has no effect on standard, non-secure connections.
&H00400	ftpFeatureCCC	The server supports the CCC command which returns the command channel to a non-secure mode. Changing this feature has no effect on standard, non-secure connections.
&H00800	ftpFeatureHOST	The server supports the HOST command which enables a client to specify the hostname after establishing a connection with a server that supports virtual hosting.
&H01000	ftpFeatureMLST	The server supports the MLST command which returns status information for files. If this feature is enabled, the MLST command will be used instead of the STAT command.
&H02000	ftpFeatureMFMT	The server supports the MFMT command which is used to change the last modification time for a file. If this command is supported, it is used instead of the MDTM command to change the modification time for a file.
&H04000	ftpFeatureXCRC	The server supports the XCRC command which returns the CRC-32 checksum for the contents of a specified file. This command is used for file verification.
&H08000	ftpFeatureMD5	The server supports the XMD5 command which returns an MD5 hash for the contents of a specified file. This command is used for file verification.
&H10000	ftpFeatureLANG	The server supports the LANG command which sets the language used for the current client session. Command responses and file naming conventions will use the specified language.
&H20000	ftpFeatureUTF8	The server supports the OPTS UTF-8 command which specifies UTF-8 encoding when specifying filenames. This feature is typically used in conjunction with setting the default language for the client session.
&H40000	ftpFeatureXQUOTA	The server supports the XQUOTA command which returns quota information for the current client session.
&H80000	ftpFeatureUTIME	The server supports the UTIME command which is used to change the last modification time for a specified file.

When a client connection is first established, all features are enabled by default. However, as the client issues commands to the server, if the server reports that the command is unrecognized that feature will automatically be disabled in the client.

For example, the first time an application calls the **GetFileSize** method to determine the size of a file, the control will try to use the SIZE command. If the server reports that the SIZE command is not available, that feature will be disabled and the control will not use the command again during the session unless it is explicitly re-enabled. This is designed to prevent the control from repeatedly sending invalid commands to a server, which may result in the server aborting the connection.

Setting the **Features** property enables those features which have been specified. More than one feature may be enabled by combining the above constants using a bitwise Or operator. To test if a particular feature has been enabled, use the bitwise And operator. For example, in Visual Basic this can be done using the And and Or operators:

```
' If the SIZE command is enabled, disable it and make sure
' that the STAT command is enabled instead
If (FileTransfer1.Features And ftpFeatureSize) <> 0 Then
    FileTransfer1.Features = FileTransfer1.Features And Not ftpFeatureSize
    FileTransfer1.Features = FileTransfer1.Features Or ftpFeatureStat
End If
```

Because features are specific to the current session, once you disconnect from the server they are reset. Even if you wish to reconnect to the same server, you must explicitly set the **Features** property again to those features which you wish to enable. Setting the **Features** property when the control is not connected to a server will cause the client session to only use those specified features for the next connection that is established. Setting the **Features** property during an active connection will change the features available for that session.

Data Type

Integer (Int32)

See Also

ActivePort Property, Connect Method, Disconnect Method

FileType Property

Gets and sets the current file transfer type for FTP transfers.

Syntax

object.**FileType** [= *filetype*]

Remarks

The **FileType** property specifies the type of file transfer between the local and server. The file transfer types supported are:

Value	Description
fileTypeAuto	The file type should be automatically determined based on the file name extension. If the file extension is unknown, the file type should be determined based on the contents of the file. The control has an internal list of common text file extensions, and additional file extensions can be registered using the **AddFileType** method.
fileTypeASCII	The file being transferred is an ASCII text file. The characters the mark the end of a line (for example, a carriage return/linefeed pair under MS-DOS) are automatically converted to the format used by the target operating system. The constant **fileTypeText** is an alias for this value.
fileTypeEBCDIC	The file being transferred is a text file created using the EBCDIC character set. If a file is being uploaded, ASCII characters are automatically converted to EBCDIC. If the file is being downloaded, EBCDIC characters are automatically converted to ASCII.
fileTypeImage	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executable programs. If the type of file cannot be automatically determined, it will always be considered a binary file. If this file type is specified when uploading or downloading text files, the native end-of-line character sequences will be preserved. The constant **fileTypeBinary** is an alias for this value.

Data Type

Integer (Int32)

See Also

AddFileType Method, GetFile Method, PutFile Method

Fingerprint Property

Returns a string that uniquely identifies the server.

Syntax

object.**Fingerprint**

Remarks

The **Fingerprint** property returns a string that consists of a series of hexadecimal values separated by colons. The value is unique to the server, and is an MD5 hash of the RSA host key. An application can use this value to determine if a connection has been established with the server previously by storing the server's host name, IP address and fingerprint in a file, registry key or a database.

Note that this property only returns a meaningful value after a secure connection has been established using the SSH protocol. For all other connections, it will return an empty string.

Data Type

String

See Also

Connect 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

IsBlocked Property

Determine 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 can be used to ensure that another blocking operation is not in progress at the time.

If this 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 the control is already performing a blocking operation.

Data Type

Boolean

See Also

LastError Property

IsConnected Property

Determine if the control is connected to a server.

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

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

KeepAlive Property

Enable monitoring of the command channel to keep the client session active.

Syntax

object.**KeepAlive** [= { True | False }]

Remarks

Setting the **KeepAlive** property to a value of true specifies that a background worker thread will be created to monitor the command channel for an FTP connection and periodically send NOOP commands to the server if no commands have been sent recently. This can prevent the server from terminating the client connection during idle periods where no commands are being issued. However, it is important to keep in mind that many servers can be configured to also limit the total amount of time a client can be connected to the server, as well as the amount of time permitted between file transfers. If the server does not respond to the NOOP command, this option will be automatically disabled for the remainder of the client session.

It is recommended that you only enable this option if the connection to the FTP server must be maintained for a relatively long period of time where there will be periods of inactivity. Never make the assumption that this option can prevent the server from terminating the connection. Most sites, particularly public FTP servers accessed over the Internet, have fairly restrictive policies designed to prevent clients from maintaining long-term connections. In most cases, if there are periods of time where your application will not be transferring files, it is more appropriate to disconnect from the server and then reconnect at a later point rather than attempting to hold the connection open.

The default value for this property is false. This property is only meaningful for FTP connections.

Data Type

Boolean

See Also

Connect Method, GetFile Method, PutFile Method

LastError Property

Gets and sets the last error code.

Syntax

object.**LastError** [= *lasterror*]

Remarks

The **LastError** property can be read to determine the last error that occurred for this instance of the object. If a value is assigned to this property, it must either be zero (to clear the error) or a valid error code.

Data Type

Integer (Int32)

See Also

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

LocalAddress Property

Return the Internet address of the local host.

Syntax

object.**LocalAddress**

Remarks

The **LocalAddress** property returns the Internet address of the local host as a string in dotted notation. If there is an active connection to a server, then the return value will depend on the network interface that was used to establish the connection. If there isn't a connection, then the default address for the local host will be returned.

Data Type

String

See Also

LocalName Property

Localize Property

Determines if remote file dates are localized to the current timezone.

Syntax

object.**Localize** [= { True | False }]

Remarks

Setting the **Localize** property controls how remote file date and time values are localized when the **GetFileTime** method is called. If the property is set to True, then the file date and time will be adjusted to the current timezone. If the property is set to False, which is the default value, then the file date and time are returned as UTC (Coordinated Universal Time) values.

Data Type

Boolean

See Also

GetFileTime Method

LocalName Property

Return the Internet domain name of the local host.

Syntax

object.**LocalName**

Remarks

The **LocalName** property returns the Internet domain name for the local host. If there is an active connection to a server, then the domain name will depend on the network interface that was used to establish the connection. If there isn't a connection, then the default domain name for the local host will be returned.

Data Type

String

See Also

LocalAddress Property

Options Property

Gets and sets the options for the current object.

Syntax

object.**Options** [= *options*]

Remarks

The **Options** property returns or modifies the options used for retrieving and sending files. The value is represented as one or more bit flags which may be combined using the logical ***or*** operator. The following options are defined:

Value	Description	
fileOptionNoCache	This instructs an HTTP server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.	
&H1000	fileOptionSecureImplicit	This option specifies the client should immediately negotiate for a secure session upon establishing a connection with the server. This is the default method for connecting to a secure HTTP server and may also be used with FTP servers that accept secure connections on port 990.
&H2000	fileOptionSecureExplicit	This option specifies the client should use the AUTH server command to tell an FTP server that it wishes to explicitly negotiate a secure connection. This requires that the server support the AUTH TLS or AUTH SSL commands. Some servers may not require this option, and some may require the option only if a port other than 990 is specified. If this option is specified, the **Secure** property will automatically be set to True.
&H4000	fileOptionSecureShell	This option specifies the client should use the Secure Shell (SSH) protocol to establish the connection. This option will automatically be selected if the connection is established using port 22, the default port for SSH, and is only required if the server is configured to use a non-standard port number.
&H8000	fileOptionSecureFallback	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.

&H40000	fileOptionPreferIPv6	This option specifies the client should only attempt to resolve a domain name to an IPv6 address. If the domain name has both an IPv4 and IPv6 address assigned to it, the default is to use the IPv4 address for compatibility purposes. Enabling this option forces the client to always use the IPv6 address if one is available. If the domain name does not have an assigned IPv4 address, the IPv6 address will always be used regardless if this option is specified.
&H100000	fileOptionHiResTimer	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
&H200000	fileOptionTLSReuse	This option specifies that TLS session reuse should be enabled for secure data connections. Some servers may require this option be enabled, although it should only used when required. This option is only valid for secure FTP (FTPS) connections and is not used with SFTP or secure HTTP connections. See the remarks below for more information.

The **Options** property enables you to set the default options for subsequent connections using the **Connect** method, and some options may only be available for certain types of connections. For example, the **fileOptionSecureExplicit** option is only valid for secure FTP connections.

The **fileOptionTLSReuse** option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627. To avoid potential problems with server compatibility, you should not specify this option for all FTP connections. It should only be used if specifically required by the server and your end-users should have the ability to selectively enable or disable this option.

Data Type

Integer (Int32)

See Also

Secure Property, Connect Method

Passive Property

Enable or disable passive mode file transfers.

Syntax

object.**Passive** [= {True | False}]

Remarks

The **Passive** property enables or disables passive FTP file transfers between the local and server. In passive transfer mode, the client is responsible for establishing the data connection between the server and the local system. By default, the **Passive** property is set to True, with the client establishing the data connection with the server.

The majority of FTP servers support passive mode transfers, and in most cases, passive mode is required when attempting to upload or download files when the client is behind a firewall or a router that performs Network Address Translation (NAT). However, if the **Passive** property is set to True and the server does not support passive mode, an error will be returned the next time a file transfer or directory listing is attempted. In this case, set the **Passive** property to False and attempt the transfer again.

Note that setting this property has no effect when uploading or downloading a file using the Hypertext Transfer Protocol (HTTP).

Data Type

Boolean

See Also

GetFile Method, PutFile Method

Password Property

Gets and sets the password for the current user.

Syntax

object.**Password** [= *password*]

Remarks

The **Password** property specifies the password used to authenticate the user. The **UserName** and **Password** properties are used together to provide credentials to the server that will authenticate the client session. If these properties not set when the connection is established, the session will be anonymous. A server may allow a client to download files without authentication, but it is usually required when uploading files.

If you are connecting to a server using the SFTP (SSH) protocol and the server requires public/private key authentication, your application should set the **PrivateKey** property to the path of the user's private key. In this case, the **Password** property can be used to specify the password required to access the private key. If the private key was created without a password, this property should be set to an empty string.

Data Type

String

See Also

PrivateKey Property, UserName Property, Connect Method

Priority Property

Gets and sets a value which specifies the priority of file transfers.

Syntax

object.**Priority** [= *priority*]

Remarks

The **Priority** property can be used to control the processor usage, memory and network bandwidth allocated for file transfers. One of the following values may be specified:

Value	Description
filePriorityBackground	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
filePriorityLow	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
filePriorityNormal	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
filePriorityHigh	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
filePriorityCritical	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

The **filePriorityNormal** priority balances resource utilization and transfer speed while ensuring that a single-threaded application remains responsive to the user. Lower priorities reduce the overall resource utilization at the expense of transfer speed. For example, if you create a worker thread to download a file in the background and want to ensure that it has a minimal impact on the process, the **filePriorityBackground** value can be used.

Higher priority values increase the memory allocated for the transfers and increases processor utilization for the transfer. The **filePriorityCritical** priority maximizes transfer speed at the expense of system resources. It is not recommended that you increase the file transfer priority unless you understand the implications of doing so and have thoroughly tested your application. If the file transfer is being performed in the main UI thread, increasing the priority may interfere with the normal processing of Windows messages and cause the application to appear to become non-responsive. It is also important to note that when the priority is set to **filePriorityCritical**, normal progress events will not be generated during the transfer.

Data Type

Boolean

See Also

GetData Method GetFile Method PutData Method PutFile Method

PrivateKey Property

Gets and sets the private key file used for SSH authentication.

Syntax

object.**PrivateKey** [= *filename*]

Remarks

The **PrivateKey** property specifies the path to the private key file used to authenticate the user. The private key is used in combination with the value of the **UserName** property to provide credentials to an SSH server. If public/private key authentication is not required by the server, this property is should be set to an empty string. This property is only used with SFTP connections and is ignored for standard FTP connections or secure connections using FTPS (FTP+TLS).

The **PrivateKey** property value can use environment variables enclosed in percent symbols, and the path to the private key file will be normalized. It is recommended you always use an absolute path to the private key file. If you do not include a path, it will use the current working directory for the process. This can produce inconsistent results because the current working directory for a process is a global value and it can be changed at any time.

The private key file name must resolve to a text file which can be read by the current process. If the file does not exist, or it does not specify a PEM formatted file which contains an RSA or OpenSSH private key, the connection will fail and the last error code will be set to `stErrorInvalidPrivateKey`.

Data Type

String

Example

```
FileTransfer1.ServerType = fileServerFtp
FileTransfer1.Options = fileOptionSecureShell
FileTransfer1.UserName = "username"
FileTransfer1.PrivateKey = "%USERPROFILE%\\.ssh\\id_rsa.pem"

nError = FileTransfer1.Connect("file.server.tld", 22)
If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString, vbExclamation
    Exit Sub
End If
```

In this example, the path to the private key file will be expanded to an absolute path because the USERPROFILE environment variable defines the home directory for the current user.

See Also

Password Property, UserName Property, Connect Method

ProtocolVersion Property

Gets and sets the current HTTP protocol version.

Syntax

object.**ProtocolVersion** [= *protocolversion*]

Remarks

The **ProtocolVersion** property sets or returns the current HTTP version number. It is used to determine how requests are submitted to the server, as well as what header fields are required. The default value for this property is "1.0", and should be changed before any connection attempt is made by the client.

Note that setting the property value to "0.9" tells the control to use the preliminary protocol specification, circa 1994. This version of the protocol only supported a basic version of the GET command, and did not have any provisions for features such as user authentication.

Data Type

String

See Also

Connect Method

ProxyPassword Property

Gets and sets the proxy server password for the current user.

Syntax

object.**ProxyPassword** [= *proxypassword*]

Remarks

The **ProxyPassword** property specifies the password used to authenticate the user to the proxy server. If a password is not required by the server, this property is ignored.

Data Type

String

See Also

Password Property, ProxyPort Property, ProxyServer Property, ProxyType Property, ProxyUser Property, ServerName Property, UserName Property, Connect Method

ProxyPort Property

Gets and sets the port number for the proxy server.

Syntax

object.**ProxyPort** [= *proxyport*]

Remarks

The **ProxyPort** property is used to set the port number that the control will use to establish a connection with the proxy server. A value of zero specifies that the client will connect to the proxy server using the standard FTP or HTTP service port.

Data Type

Integer (Int32)

See Also

Password Property, ProxyPassword Property, ProxyServer Property, ProxyType Property, ProxyUser Property, ServerName Property, UserName Property, Connect Method

ProxyServer Property

Gets and sets the host name of the proxy server.

Syntax

object.**ProxyServer** [= *proxyserver*]

Remarks

The **ProxyServer** property should be set to the name of the proxy server that you want to connect to. This property may be set to either a fully qualified domain name, or an IP address. This property is only used if the **ProxyType** property is set to a non-zero value.

Data Type

String

See Also

Password Property, ProxyPassword Property, ProxyPort Property, ProxyType Property, ProxyUser Property, ServerName Property, UserName Property, Connect Method

ProxyType Property

Gets and sets the current proxy server type.

Syntax

object.**ProxyType** [= *proxytype*]

Remarks

FTP and HTTP proxy servers have different characteristics, as described in the following table:

Value	Description
fileProxyNone	No proxy server is being used. This is the default value.
fileProxyUser	The client is not logged into the proxy server. The USER command is sent in the format username@ftpsite followed by the password. This is the format used with the Gauntlet proxy server.
fileProxyLogin	The client is logged into the proxy server. The USER command is then sent in the format username@ftpsite followed by the password. This is the format used by the InterLock proxy server.
fileProxyOpen	The client is not logged into the proxy server. The OPEN command is sent specifying the host name, followed by the username and password.
fileProxySite	The client is logged into the server. The SITE command is sent, specifying the host name, followed by the username and the password.
fileProxyOther	This special proxy type specifies that another, undefined proxy server is being used. The client connects to the proxy host, but does not attempt to authenticate the client. The application is responsible for negotiating with the proxy server, typically using the **Command** method to send specific command sequences.
fileProxyCern	The client is connected to a proxy server; the resource path is sent as a complete URL. This proxy type is only valid for HTTP servers.
fileProxyTunnel	The client is connecting through a standard CERN proxy server to a secure web server and the resource is specified as a complete URL. The **Secure** property should be set to true when using this proxy type.
fileProxyWindows	The client should use the default proxy configuration specified for the system.

There are duplicated values for **ProxyType** because this property value is interpreted according to the protocol that is being used.

Data Type

Integer (Int32)

See Also

Password Property, ProxyPassword Property, ProxyPort Property, ProxyServer Property, ProxyUser Property, ServerName Property, UserName Property, Connect Method

ProxyUser Property

Gets and sets the current proxy user name.

Syntax

object.**ProxyUser** [= *proxyuser*]

Remarks

The **ProxyUser** property specifies the user that is logging in to the proxy server. If the proxy server does not require the user to login, then this property is ignored.

Data Type

String

See Also

Password Property, ProxyPassword Property, ProxyPort Property, ProxyServer Property, ProxyType Property, ServerName Property, UserName Property, Connect Method

Resource Property

Gets and sets the name of a resource on the server.

Syntax

object.**Resource** [= *value*]

Remarks

The **Resource** property is used to specify the name of a resource on the server. The resource may be a file, such as an HTML document or an image, or it may be a script used to process data submitted by the client. Note that this property specifies the name of the resource only, not a complete URL. To specify a complete URL, set the **URL** property and the control will automatically set the **Resource** property to the correct value.

Data Type

String

See Also

URL Property, Connect Method, GetFile Method, PutFile Method

ResultCode Property

Return the result code of the previous action.

Syntax

object.**ResultCode**

Remarks

The **ResultCode** read-only property returns the result code of the last action performed by the client. Result codes are either Boolean (in which a zero value indicates an error and a non-zero value indicates success), or are three-digit numeric values returned by the server. If the result codes fall into the later category, they may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

Data Type

Integer (Int32)

See Also

ResultString Property

ResultString Property

Return a string describing the results of the previous action.

Syntax

object.**ResultString**

Remarks

The **ResultString** read-only property returns the result string from the last action taken by the client. This string is generated by the server, and typically is used to describe the result code. For example, if an error is indicated by the result code, the result string may describe the condition that caused the error.

Data Type

String

See Also

ResultCode Property

Secure Property

Specify 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 must set this property value to True prior to calling the **Connect** method. Once the connection has been established, the client may retrieve messages from the server as with standard connections.

It is strongly recommended that any application that sets this property to True use error handling to trap any errors that may occur. If the control 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.

If you are connecting to an FTP server, you should check to see if the server requires the use of the AUTH command to establish an explicit, secure session. If so, you must set the **Options** property to **fileOptionSecureExplicit** prior to calling the Connect method.

Data Type

Boolean

Example

The following example establishes a secure connection to a server and retrieves a file:

```
FileTransfer1.ServerType = fileServerHttp
FileTransfer1.ServerName = strHostName
FileTransfer1.ServerPort = 443
FileTransfer1.Secure = True

nError = FileTransfer1.Connect()
If nError > 0 Then
    MsgBox "Unable to connect to server " & strHostName, vbExclamation
    Exit Sub
End If

If FileTransfer1.CertificateStatus <> stCertificateValid Then
    lResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                     "Are you sure you wish to continue?", vbYesNo)

    If lResult = vbNo Then
        FileTransfer1.Disconnect
        Exit Sub
    End If
End If

nError = FileTransfer1.GetFile(strLocalFile, strRemoteFile)
FileTransfer1.Disconnect

If nError > 0 Then
    MsgBox "Unable to retrieve file from server " & strHostName
    Exit Sub
End If
```

See Also

URL 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
stCipherNone	No cipher has been selected. This is not a secure connection with the server.
stCipherRC2	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.
stCipherRC4	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.
stCipherRC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
stCipherDES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher using 56-bit keys.
stCipherDES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively using a 168-bit key length.
stCipherDESX	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.
stCipherAES	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.
stCipherSkipjack	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
stCipherBlowfish	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
stHashMD5	The MD5 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA1	The SHA-1 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA256	The SHA-256 algorithm has been selected.
stHashSHA384	The SHA-384 algorithm has been selected.
stHashSHA512	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
stKeyExchangeNone	No key exchange algorithm has been selected. This is not a secure connection with the server.
stKeyExchangeRSA	The RSA public key exchange algorithm has been selected.
stKeyExchangeKEA	The KEA public key exchange algorithm has been selected. This is an improved version of the Diffie-Hellman public key algorithm.
stKeyExchangeDH	The Diffie-Hellman public key exchange algorithm has been selected.
stKeyExchangeECDH	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

ServerDirectory Property

Gets and sets the current working directory on the FTP server.

Syntax

object.**ServerDirectory** [= *serverdirectory*]

Remarks

The **ServerDirectory** property specifies the name of a directory on the server. When a connection is first established with the FTP server, this property is set to the current working directory. Setting this property is equivalent to using the **ChangeDirectory** method.

Data Type

String

See Also

ChangeDirectory Method

ServerName Property

Gets and sets the host name for the FTP or HTTP server.

Syntax

object.**ServerName** [= *servername*]

Remarks

If the **ServerName** property is not explicitly set, then an application must provide the host name or address to the **Connect** method. Once the connection has been established, this property will be updated with the appropriate value. If the server uses a non-standard port number, it can be specified using the **ServerPort** property.

The IP address of a server may be used as the **ServerName**.

Data Type

String

See Also

ServerPort Property, ServerType Property, URL Property, Connect Method

ServerPort Property

Gets and sets the port number for a remote connection.

Syntax

object.**ServerPort** [= *serverport*]

Remarks

If the **ServerPort** property is 0, then the port to be used will be inferred from the **ServerType** and **Secure** properties.

Conversely, if the **ServerPort** is set to one of the standard FTP ports (21 or 990) or standard HTTP ports (80 or 443), and **ServerType** is undefined, then the **ServerType** and **Secure** properties will be inferred from the **ServerPort**.

The **ServerPort** property will be used by the **Connect** method if the ***ServerPort*** parameter for **Connect** is missing.

Data Type

Integer (Int32)

See Also

Secure Property, ServerName Property, ServerType Property, URL Property, Connect Method

ServerType Property

Gets and sets the type of server the control is connecting to.

Syntax

object.**ServerType** [= *servertype*]

Remarks

The **ServerType** property is used to specify the type of server the control will connect to. If this property is not explicitly set in code, then the control will attempt to automatically determine the correct server type based on the values of the **ServerPort** and **Secure** properties.

This property may return one of the following values:

Value	Description
fileServerUndefined	Server type has not been set, and may be inferred from the ServerPort and Secure properties at connection time.
fileServerFtp	Connection to a FTP server is desired, or has been achieved.
fileServerHttp	Connection to a HTTP server is desired, or has been achieved

Note that many properties and some methods are specific to a server type. Attempting to use such properties or methods while connected to a server of a type that is not supported for that operation will result in an error.

Data Type

Integer (Int32)

See Also

Secure Property, ServerName Property, ServerPort Property, URL Property, Connect Method

System Property

Return information about the server.

Syntax

object.**System** [= *system*]

Remarks

The **System** property contains information about the server operating system. Currently, this is only available through FTP servers.

Data Type

String

TaskCount Property

Return the number of active background file transfers.

Syntax

object.**TaskCount**

Remarks

The **TaskCount** property returns the number of background file transfers that are currently in progress. One common use for this property is to create a timer that periodically checks this value when a series of background transfers are started. When the property returns a value of zero, that indicates all of the background transfers have completed. This property can also be used to enumerate the active background tasks in conjunction with the **TaskList** property.

Data Type

Integer (Int32)

See Also

TaskId Property, TaskList Property, AsyncGetFile Method, AsyncPutFile Method, TaskAbort Method, TaskWait Method

TaskId Property

Return the task ID for the last background file transfer.

Syntax

object.**TaskId**

Remarks

The **TaskId** property returns the task ID associated with the last background task that started. The value of this property is only meaningful after the **AsyncGetFile** or **AsyncPutFile** method is called to initiate a background file transfer, and the value will change with each subsequent background transfer that is performed. If this property returns a value of zero, that indicates that no background tasks have been started for this instance of the control.

To enumerate the active background tasks, use the **TaskCount** property and the **TaskList** property array.

Data Type

Integer (Int32)

See Also

TaskCount Property, TaskList Property, AsyncGetFile Method, AsyncPutFile Method, TaskAbort Method, TaskWait Method

TaskList Property

Return the task ID for an active background file transfer.

Syntax

object.**TaskList**(*Index*)

Remarks

The **TaskList** property is a zero-based array that returns an ID associated with an active background task. The current number of active tasks can be determined using the **TaskCount** property. If the index value specified for this property array exceeds the number of active tasks, an exception will be thrown.

As background tasks complete and additional tasks are started, the values returned by this property array will change. The application should never make an assumption about the actual task ID values returned or the order they are returned. While task IDs are assigned sequentially, they should be considered opaque values that are unique to the process. When a background task completes, its corresponding task ID is removed from the list of active tasks and this can potentially change the task ID values associated with each index into the property array.

Data Type

Integer (Int32)

See Also

TaskCount Property, TaskId Property, AsyncGetFile Method, AsyncPutFile Method, TaskAbort Method, TaskWait Method

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** [= { True | False }]

Remarks

Error handling for 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, methods will not raise an exception if an error occurs. Instead, the application should check the return value of the method and report any errors based on that value. It is the responsibility of the application to interpret the error code and take an appropriate action. This is the default value for the property.

If the **ThrowError** property is set to True, any method which generates an error will cause the component to raise an exception which must be handled or the application will terminate.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of this property. This property only controls how errors are handled when calling methods.

Data Type

Boolean

See Also

LastError Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking network operation is aborted.

Syntax

object.**Timeout** [= *timeout*]

Remarks

The **Timeout** property controls the amount of time that the component will wait for a network operation to complete before aborting the operation and returning an error. The default value for this property is 20 seconds. It may be required to increase this value if a slow or unreliable network connection is being used.

Data Type

Integer (Int32)

Trace Property

Enable or disable network function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the tracing of network function calls and is primarily used as a debugging tool. 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 enabled or disabled for an entire process. This means that once tracing is enabled for a given instance of the object, all of the function calls made by the process will be logged.

If tracing 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 trace file is fairly large. 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.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Return or specify the network function trace output file.

Syntax

object.**TraceFile** [= *filename*]

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, then a file named CSTRACE.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 function tracing is enabled per-process, the trace file is shared by all instances of the object being used. 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:

```
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 included in brackets.

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". Those functions which expect Internet 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the current network function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

Remarks

The **TraceFlags** property is used to specify the type of information written to the trace file when network function tracing is enabled. The following values may be used:

Value	Description
fileTraceInfo	All function calls are written to the trace file. This is the default value.
fileTraceError	Only those function calls which fail are recorded in the trace file.
fileTraceWarning	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file.
fileTraceHexDump	All function calls are written to the trace file, plus all the data that is sent or received is logged, in both ASCII and hexadecimal format.

Since network function tracing is enabled per-process, the trace flags are shared by all instances of the object being used.

Warnings are generated when a non-fatal error is returned by a network function. For example, if data is being sent to the server and the error 10035 is returned, a warning is generated 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

TransferBytes Property

Return the number of bytes transferred from the server.

Syntax

object.**TransferBytes**

Remarks

The **TransferBytes** property returns the number of bytes that have been copied to or from the server. If this property is read while a transfer is ongoing, the property returns the number of bytes that have been copied up to that point. If read after a transfer has completed, the total number of bytes copied is returned.

If the value would exceed 2,147,483,647 bytes (the maximum value for a 32-bit integer) this property will return -1 to indicate an overflow condition. If you are potentially transferring files larger than 2 GiB in size, you should use the **TransferBytesXL** property instead, which returns the number of bytes as a **Double** floating-point value.

This property value is reset with every data transfer.

Data Type

Integer (Int32)

See Also

TransferBytesXL Property, TransferRate Property, TransferTime Property, OnProgress Event

TransferBytesXL Property

Return the number of bytes transferred from the server.

Syntax

object.**TransferBytesXL**

Remarks

The **TransferBytesXL** property returns the number of bytes that have been copied to or from the FTP server. This property returns the number of bytes as a **Double** floating-point value instead of a **Long** integer, making it suitable for very large files that exceed 2 GiB in size.

If this property is read while a transfer is ongoing, the property returns the number of bytes that have been copied up to that point. If read after a transfer has completed, the total number of bytes copied is returned.

This property value is reset with every data transfer.

Data Type

Double

See Also

TransferRate Property, TransferTime Property, OnProgress Event

TransferRate Property

Return the current file transfer rate in bytes per second.

Syntax

object.**TransferRate**

Remarks

The **TransferRate** property returns the rate at which the file data is being transferred, expressed in bytes per second. If this property is read while a transfer is ongoing, it returns the current average transfer rate.

If this property is read after the transfer has completed, it returns the final transfer rate which is calculated as the total number of bytes transferred divided by the number of seconds to complete the transfer. This property value is reset with every data transfer.

Data Type

Integer (Int32)

See Also

TransferBytes Property, TransferTime Property, OnProgress Event

TransferTime Property

Return the number of seconds elapsed during a data transfer.

Syntax

object.**TransferTime**

Remarks

The **TransferTime** property returns the number of seconds that have elapsed since the last data connection was opened on the server. If the property is read while a transfer is ongoing, it returns the current elapsed time since the file transfer started.

If the property is read after the transfer has completed, it returns the total number of seconds it took to transfer the file. This property value is reset with every data transfer.

Data Type

Integer (Int32)

See Also

TransferBytes Property, TransferRate Property, OnProgress Event

URL Property

Gets and sets the current URL used to access a resource on the server.

Syntax

object.**URL** [= *url*]

Remarks

The **URL** property returns the current Uniform Resource Locator string which is used by the control to access a resource on the server. URLs have a specific format which provides information about the server, port, resource, as well as optional information such as a username and password for authentication:

```
[ftp|ftps|sftp]://[username:
[password]@]hostname[:port]/[path/...]filename[;type=id]
```

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

The first part of the URL is the scheme and in this case will always be "ftp", "ftps", "sftp", "http" or "https" depending on the protocol that is required. If a username and password is required for authentication, then this will be included in the URL before the name of the server. Next, there is the name of the server to connect to, optionally followed by a port number. If no port number is given, then the default port for the protocol will be used. This is followed by the resource, which is usually a path to a file or script on the server. Parameters to the resource may also be specified, which are typically used as arguments to a script that is executed on the server.

Here are some common examples of URLs used to access resources on an file server:

`ftp://www.example.com/pub/financial/jan2023.xlsx`
In this example, the server is www.example.com, the path is "pub/financial" and the file name is "jan2023.xlsx". The default port will be used to access the file, and no username and password is provided for authentication so this file must be publicly available to anonymous users.

`ftps://executive:secret@www.example.com/corporate/projections/sales2024.xlsx`

In this example, the server is www.example.com and, the path is "corporate/projections" and the file name is "sales2024.xlsx". Because the protocol is ftps, a secure connection on port 990 will be established. The user name "executive" and password "secret" will be used to authenticate the session.

`http://www.example.com/products/index.html`
In this example, the server is www.example.com and the resource is /products/index.html. The default port will be used to access the resource, and no username and password is provided for authentication.

`https://www.example.com/order/confirm.asp`
In this example, the server is www.example.com and the resource is the script /order/confirm.asp. Because the protocol is https, a secure connection on port 443 will be established.

When setting the **URL** property, the control will parse the string and automatically update the **ServerName**, **ServerPort**, **UserName**, **Password** and **Resource** properties according to the values specified in the URL. This enables an application to simply provide the URL and then call the **Connect**

method to establish the connection.

Note that if this property is assigned a value which cannot be parsed, the control will throw an error that indicates that the property value is invalid. In a language like Visual Basic it is important that you implement an error handler, particularly if you are assigning a value to the property based on user input. If the user enters an invalid URL and there is no error handler, it could result in an exception which terminates the application.

Data Type

String

Example

```
' Setup error handling since the control will throw an error
' if an invalid URL is specified

On Error Resume Next: Err.Clear
FileTransfer1.URL = Text1.Text

' Check the Err object to see if an error has occurred, and
' if so, let the user know that the URL is invalid

If Err.Number <> 0 Then
    MsgBox "The specified URL is invalid", vbExclamation
    Text1.SetFocus
    Exit Sub
End If

' Reset error handling and connect to the server using the
' default property values that were updated when the URL
' property was set (ie: ServerName, ServerPort, Resource, etc.)

On Error GoTo 0
nError = FileTransfer1.Connect()

If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString, vbExclamation
    Exit Sub
End If

' Download the resource and store it in the specified file
nError = FileTransfer1.GetFile(strLocalFile)
```

See Also

Password Property, Resource Property, ServerName Property, ServerPort Property, ServerType Property, UserName Property, Connect Method

UserName Property

Gets and sets the current user name.

Syntax

object.**UserName** [= *username*]

Remarks

The **UserName** property specifies the name used to authenticate the user. If the property is not explicitly set, then an application may provide the user name to the **Connect** method. Once the connection has been established, this property will be updated with the appropriate value.

The **UserName** and **Password** properties are typically required for HTTP uploads and FTP connections. They are typically not required for HTTP downloads. If the **UserName** and **Password** properties are undefined for an FTP connection attempt, then anonymous FTP will be used.

Data Type

String

See Also

Password Property, PrivateKey Property, Connect Method

Version Property

Return the current version of the object.

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

File Transfer Control Methods

Method	Description
AddFileType	Associate a file name extension with a specific file type
AsyncGetFile	Download a file from the server to the local system in the background
AsyncPutFile	Upload a file from the local system to the server in the background
Cancel	Cancels the current blocking network operation
ChangeDirectory	Changes current directory on remote FTP server
CloseDirectory	Close the directory that was opened for reading on the FTP server
Command	Specify a command to execute on a FTP server
Connect	Establish a connection with the specified FTP or HTTP server
DeleteFile	Remove a file on the server
Disconnect	Disconnect from the FTP or HTTP server
GetData	Transfers data from a file on the server and store it in a local buffer
GetDirectory	Return the current working directory on the server
GetFile	Download a file from the server to the local system
GetFileList	Return an unparsed list of files in the specified directory
GetFilePermissions	Return the access permissions for a file on the server
GetFileSize	Returns the size of the specified file on the server
GetFileStatus	Return status information about a specific file on an FTP server
GetFileTime	Returns the modification date and time for specified file on the server
GetFileType	Return the default file type for the current session
GetMultipleFiles	Download multiple files from the server to the local system
GetText	Download the contents of a text file to a string buffer
Initialize	Initialize the component and load the networking library
MakeDirectory	Create a new directory on the remote FTP host
OpenDirectory	Open the specified directory on an FTP server for reading
PostFile	Upload a file from the local system to a script on a web server
PutData	Transfers data from a local buffer and stores it in a file on the server
PutFile	Upload a file from the local system to the server
PutMultipleFiles	Upload multiple files from the local system to the remote system
PutText	Create a text file on the server from the contents of a string buffer
ReadDirectory	Read a directory entry from an FTP server
RemoveDirectory	Remove a directory on the remote FTP server

RenameFile	Change the name of an existing file on the FTP server
Reset	Reset the internal state of the control
SetFilePermissions	Change the access permissions for a file on the server
SetFileTime	Changes the modification date and time for a file on the server
TaskAbort	Abort the specified asynchronous task
TaskDone	Determine if an asynchronous task has completed
TaskResume	Resume execution of an asynchronous task
TaskSuspend	Suspend execution of an asynchronous task
TaskWait	Wait for an asynchronous task to complete
Uninitialize	Uninitialize the component and unload the networking library
VerifyFile	Compare the contents of a local file against a file stored on the server

AddFileType Method

Associate a file name extension with a specific file type.

Syntax

object.**AddFileType**(*FileExtension*, *FileType*)

Parameters

FileExtension

A string that specifies the file name extension.

FileType

Specifies the type of file associated with the file extension. This parameter can be one of the following values:

Value	Description
fileTypeText	The file being transferred is an ASCII text file. The characters the mark the end of a line (for example, a carriage return/linefeed pair under MS-DOS) are automatically converted to the format used by the target operating system.
fileTypeEBCDIC	The file being transferred is a text file created using the EBCDIC character set. If a file is being uploaded, ASCII characters are automatically converted to EBCDIC. If the file is being downloaded, EBCDIC characters are automatically converted to ASCII.
fileTypeBinary	The file is transferred without any modification. This is the default file transfer type, and should be used when transferring binary (non-text) data.

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 **AddFileType** method is used to associate specific file types with file name extensions. The control has an internal list of standard text file extensions which it automatically recognizes. This method can be used to extend or modify that list for the client session.

See Also

Features Property, FileType Property, GetFileStatus Method, GetFileTime Method

AsyncGetFile Method

Download a file from the server to the local system in the background.

Syntax

object.**AsyncGetFile**(*LocalFile*, *RemoteFile*, [*Offset*])

Parameters

LocalFile

A string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

RemoteFile

A string that specifies the file on the server that will be transferred to the local system. The file pathing and name conventions must be that of the server.

Offset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully.

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 **AsyncGetFile** method will download the contents of a remote file to a file on the local system. It is similar to the **GetFile** method, however it retrieves the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being downloaded from the server. This method requires that you explicitly establish a connection using the **Connect** method. All background tasks will duplicate the active connection and use it establish a secondary connection with the server to perform the file transfer. If you wish to perform multiple asynchronous file transfers from different servers, you must create an instance of the control for each server.

After this method is called, the **OnTaskBegin** event will be fired, indicating that the background task has begun the process of connecting to the server and performing the file transfer. As the file is downloaded, the control will periodically invoke the **OnTaskRun** event handler. When the transfer has completed, the **OnTaskEnd** event will be fired. It is not required that you implement handlers for these events.

To determine when a transfer has completed without implementing any event handlers, periodically call the **TaskDone** method. If you wish to block the current thread and wait for the transfer to complete, call the **TaskWait** method. To stop a background file transfer that is in progress, call the **TaskAbort** method. This will signal the background worker thread to cancel the transfer and terminate the session.

This method can be called multiple times to download more than one file in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. The application should not make any assumptions about the sequence in which background transfers are performed or the order in which they may complete.

Example

```
' Establish a connection to the server
nError = FileTransfer1.Connect(strHostName, 21, strUserName, strPassword)

If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString, vbExclamation
    Exit Sub
End If

' Download a file in the background
nError = FileTransfer1.AsyncGetFile(strLocalFile, strRemoteFile)

If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

TaskId Property, AsyncPutFile Method, TaskAbort Method, TaskDone Method, TaskWait Method, OnTaskBegin Event, OnTaskEnd Event, OnTaskRun Event

AsyncPutFile Method

Upload a file from the local system to the server in the background.

Syntax

object.**AsyncPutFile**(*LocalFile*, *RemoteFile*, [*Options*], [*Offset*])

Parameters

LocalFile

A string that specifies the file on the local system that will be transferred to the server. The file pathing and name conventions must be that of the local host.

RemoteFile

A string that specifies the file on the server that will be created, overwritten or appended to. The file pathing and name conventions must be that of the server.

Offset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully.

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 **AsyncPutFile** method will upload the contents of a file on the local system to the server. It is similar to the **PutFile** method, however it retrieves the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being uploaded to the server. This method requires that you explicitly establish a connection using the **Connect** method. All background tasks will duplicate the active connection and use it establish a secondary connection with the server to perform the file transfer. If you wish to perform multiple asynchronous file transfers from different servers, you must create an instance of the control for each server.

After this method is called, the **OnTaskBegin** event will be fired, indicating that the background task has begun the process of connecting to the server and performing the file transfer. As the file is uploaded, the control will periodically invoke the **OnTaskRun** event handler. When the transfer has completed, the **OnTaskEnd** event will be fired. It is not required that you implement handlers for these events.

To determine when a transfer has completed without implementing any event handlers, periodically call the **TaskDone** method. If you wish to block the current thread and wait for the transfer to complete, call the **TaskWait** method. To stop a background file transfer that is in progress, call the **TaskAbort** method. This will signal the background worker thread to cancel the transfer and terminate the session.

This method can be called multiple times to upload more than one file in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. The application should not make any assumptions about the sequence in which background transfers are performed or the order in which they may complete.

Example

```
' Establish a connection to the server
nError = FileTransfer1.Connect(strHostName, 21, strUserName, strPassword)

If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString, vbExclamation
    Exit Sub
End If

' Upload a file in the background
nError = FileTransfer1.AsyncPutFile(strLocalFile, strRemoteFile)

If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

TaskId Property, AsyncGetFile Method, TaskAbort Method, TaskDone Method, TaskWait Method, OnTaskBegin Event, OnTaskEnd Event, OnTaskRun Event

Cancel Method

Cancels the current blocking network operation.

Syntax

object.**Cancel**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

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.

Example

```
Private Sub cmdCancel_Click()
    Dim nError As Long

    nError = FileTransfer1.Cancel()
    If nError > 0 Then
        MsgBox "Cancel error: " & nError
    End If
End Sub
```

See Also

Disconnect Method, Reset Method, OnCancel Event

ChangeDirectory Method

Changes current directory on remote FTP server.

Syntax

object.**ChangeDirectory**(*Path*)

Parameters

Path

A string value which specifies the directory on the server.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ChangeDirectory** method changes the current working directory on the server. This method updates the **ServerDirectory** property to reflect the new path.

Example

```
Private Sub cmdChgDir_Click()
    Dim nError As Long

    nError = FileTransfer1.ChangeDirectory(Trim(txtServerDirectory.Text))
    If nError > 0 Then
        MsgBox "ChangeDirectory error: " & nError
    End If
    txtServerDirectory.Text = FileTransfer1.ServerDirectory
End Sub
```

See Also

ServerDirectory Property

CloseDirectory Method

Close the directory that was opened for reading on the FTP server.

Syntax

object.**CloseDirectory**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CloseDirectory** method closes the directory that was opened on the server using the **OpenDirectory** method. This method must be called once all of the files have been read from the server, otherwise an error will be returned on all subsequent attempts to transfer files or read other directories.

See Also

OpenDirectory Method, ReadDirectory Method

Command Method

Specify a command to execute on a FTP server.

Syntax

object.**Command**(*Command*)

Parameters

Command

A string value which specifies the command that will be executed on the server.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

Invoking the **Command** method causes the specified command to be executed on the server. This allows the application to take advantage of extended commands that are not directly available through the control. The **ResultCode** property should be checked to determine if the command was successful or not.

Example

```
Dim nError As Long

' Report current status of the server
nError = FileTransfer1.Command("STAT")

If nError > 0 Then
    MsgBox "Command error: " & nError
Else
    MsgBox FileTransfer1.ResultCode & ": " & FileTransfer1.ResultString
End If
```

See Also

ResultCode Property, ResultString Property, OnCommand Event

Connect Method

Establish a connection with the specified FTP or HTTP server.

Syntax

object.**Connect**([*ServerName*], [*ServerPort*], [*UserName*], [*Password*], [*Timeout*], [*Options*])

Parameters

ServerName

An optional string value which specifies the host name or IP address of the server. If this argument is not specified, it defaults to the value of the **ServerName** property if it is defined.

ServerPort

A number which specifies the port to connect to on the server. If this argument is not specified, it defaults to the value of the **ServerPort** property. A value of zero indicates that the default port number for the service type should be used to establish the connection.

UserName

A string which specifies the user name which will be used to authenticate the client session. This value must specify a valid user name and cannot be an empty string. If this argument is not specified, it defaults to the value of **UserName** property.

Password

A string which specifies the password which will be used to authenticate the client session. If the user does not have a password, this value can be an empty string. If this argument is not specified, it defaults to the value of the **Password** property. If the server requires public/private key authentication, set the **PrivateKey** property to the name of the file containing the private key prior to calling this method.

Timeout

The number of seconds that the client will wait for a response before failing the operation. If this argument is not specified, the value of the **Timeout** property will be used as the default.

Options

An optional integer value which specifies one or more options. If this argument is omitted or a value of zero is specified, a default connection will be established. This argument is constructed by using a bitwise operator with any of the following values

Value	Description
fileOptionNoCache	This instructs an HTTP server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
&H1000	fileOptionSecureImplicit	This option specifies the client should immediately negotiate for a secure session upon establishing a connection with the server. This is the default method for

		connecting to a secure HTTP server and may also be used with FTP servers that accept secure connections on port 990.
&H2000	fileOptionSecureExplicit	This option specifies the client should use the AUTH server command to tell an FTP server that it wishes to explicitly negotiate a secure connection. This requires that the server support the AUTH TLS or AUTH SSL commands. Some servers may not require this option, and some may require the option only if a port other than 990 is specified. If this option is specified, the **Secure** property will automatically be set to True.
&H4000	fileOptionSecureShell	This option specifies the client should use the Secure Shell (SSH) protocol to establish the connection. This option will automatically be selected if the connection is established using port 22, the default port for SSH, and is only required if the server is configured to use a non-standard port number.
&H8000	fileOptionSecureFallback	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.
&H40000	fileOptionPreferIPv6	This option specifies the client should only attempt to resolve a domain name to an IPv6 address. If the domain name has both an IPv4 and IPv6 address assigned to it, the default is to use the IPv4 address for compatibility purposes. Enabling this option forces the client to always use the IPv6 address if one is available. If the domain name does not have an assigned IPv4 address, the IPv6 address will always be used regardless if this option is specified.
&H100000	fileOptionHiResTimer	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.

&H200000	fileOptionTLSReuse	This option specifies that TLS session reuse should be enabled for secure data connections. Some servers may require this option be enabled, although it should only used when required. This option is only valid for secure FTP (FTPS) connections and is not used with SFTP or secure HTTP connections. See the remarks below for more information.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Connect** method is used to establish a connection with the specified server. This is the first method that must be called prior to the application transferring files or issuing FTP commands. If the **Connect** method is called when a connection already exists, the current connection will be closed.

It is permissible to specify a complete URL as the first argument to the method and the connection will be established with the server using specified protocol. Passing a complete URL to the **Connect** method has the same effect as setting the **URL** property and then calling the method with no arguments.

Additional properties that affect the operation of the **Connect** method are:

- KeepAlive Property (HTTP)
- ProtocolVersion Property (HTTP)
- ProxyPassword Property
- ProxyPort Property
- ProxyServer Property
- ProxyType Property
- ProxyUser Property
- Secure Property
- ServerType Property
- URL Property

If the **ServerType** property has the value ***fileServerUndefined***, then the **Connect** method will try to infer the server type from the value of the **ServerPort** property. If the server type cannot be automatically determined, an error will be returned and the server type must be explicitly specified.

The **fileOptionTLSReuse** option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627. To avoid potential problems with server compatibility, you should not specify this option for all FTP connections. It should only be used if specifically required by the FTP server and your end-users should have the ability to selectively enable or disable this option.

This method will return a value of zero if the action was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Example

```
' Connect using a hostname and port number
nError = FileTransfer1.Connect("www.example.com", 443)
If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString, vbExclamation
    Exit Sub
End If

' Connect using a URL
nError = FileTransfer1.Connect("https://wwww.example.com")
If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

KeepAlive Property, Options Property, Secure Property, ServerType Property, ServerName Property, ServerPort Property, UserName Property, Password Property, Timeout Property, URL Property

DeleteFile Method

Remove a file on the server.

Syntax

object.**DeleteFile**(*Filename*)

Parameters

Filename

A string value which specifies the name of the file to be deleted.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **DeleteFile** method deletes an existing file from the server. You must have the appropriate permission to delete the file, or an error will occur.

Example

```
Private Sub cmdRmFile_Click()
    Dim nError As Long

    nError = FileTransfer1.DeleteFile(Trim(txtRemoteFile.Text))
    If nError > 0 Then
        MsgBox "Delete File error: " & nError
    End If
End Sub
```

See Also

RenameFile Method

Disconnect Method

Disconnect from the FTP or HTTP server.

Syntax

object.**Disconnect**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Disconnect** method causes the network connection to be closed.

Example

```
Private Sub cmdDisconnect_Click()
    Dim nError As Long

    nError = FileTransfer1.Disconnect()
    If nError > 0 Then
        MsgBox "Disconnect error: " & nError
    End If
End Sub
```

See Also

Connect Method

GetData Method

Transfers data from a file on the server and store it in a local buffer.

Syntax

object.**GetData**(*RemoteFile*, *Buffer*, [*Length*], [*Reserved*])

Parameters

RemoteFile

A string that specifies the file on the server that will be transferred to the local system. The file pathing and name conventions must be that of the server.

Buffer

This parameter specifies the local buffer that the data will be stored in. If the variable is a String type, then the data will be returned as a string of characters. This is the most appropriate data type to use if the file on the server is a text file. If the remote file contains binary data, it is recommended that a Byte array variable be specified as the argument to this method.

Length

An optional integer argument that will contain the number of bytes copied into the buffer when the method returns.

Reserved

An argument reserved for future expansion. This argument should always be omitted or specified as a numeric value of zero.

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 **GetData** method transfers data from a file on the server to the local system, storing it in the specified buffer . This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

Priority Property, GetFile Method, PutData Method, PutFile Method, OnProgress Event

GetDirectory Method

Return the current working directory on the server.

Syntax

object.**GetDirectory**(*RemotePath*)

Parameters

RemotePath

A string variable which will contain the current working directory when the method returns. This parameter must be passed by reference.

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 **GetDirectory** method sends the PWD command to the server to get the current working directory.

See Also

ChangeDirectory, CloseDirectory Method, OpenDirectory Method, ReadDirectory Method

GetFile Method

Copy a file from the server to the local system.

Syntax

object.**GetFile**(*LocalFile*, [*RemoteFile*], [*Offset*])

Parameters

LocalFile

A string that specifies the name of the file that will be created on the local system. The file pathing and name conventions must be that of the local host.

RemoteFile

A string that specifies specifies the name of the file on the server. You must have permission to open this file for reading. This is an optional argument; if it is omitted, the value of the **Resource** property will be used. It is also permissible to specify a complete URL and the file will be downloaded from that location.

Offset

An optional integer argument that specifies the byte offset where the file transfer will begin. This argument is only valid for FTP servers, and is used to resume interrupted transfers.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **GetFile** method copies an existing file from the remote system to the local system. If the local file already exists, it is overwritten.

Not all servers support the ability to restart a file transfer. Notably, a Windows IIS server may return an error if a non-zero restart offset is specified. It is not recommended that you restart text file transfers since differences between end-of-line characters can result in byte offset differences between the local and server system.

If the **AppendFile** property is true, then the contents of ***RemoteFile*** will be appended to the contents of ***LocalFile***, if it exists. Otherwise, the contents of ***RemoteFile*** will overwrite the contents of ***LocalFile***.

Example

```
nError = FileTransfer1.GetFile(strLocalFile, strRemoteFile)
If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

AppendFile Property, URL Property, GetData Method, GetMultipleFiles Method, PostFile Method, PutData Method, PutFile Method

GetFileList Method

Return an unparsed list of files in the specified directory.

Syntax

object.**GetFileList**(*RemotePath*, *Buffer*, [*Length*], [*Options*])

Parameters

RemotePath

A string which specifies the name of a directory on the server. The list of files and subdirectories in that directory will be returned to the client. To obtain a list of files in the current working directory on the server, use an empty string.

Buffer

A buffer that the data will be stored in. It is recommend that a **String** variable type is used, although it is also possible to provide a **Byte** array as this argument, in which case the file listing will converted to ANSI characters and fill the array. Any other variable type will cause this method to throw an exception.

Length

A numeric value which specifies the maximum number of characters to read. If the argument is omitted, then the maximum size of the buffer will be calculated automatically. In most cases, it is not necessary to provide this argument.

Options

A numeric value which specifies how the list of files should be returned by the server. It may be one of the following values:

Value	Description
ftpListDefault	This option specifies the server should return a complete file list, providing all of the information available about that file. This typically includes the date and time the file was last modified, the size of the file and access rights. This option is the default, and will be used if the argument is omitted from the method call.
ftpListNameOnly	This option specifies the server should only return a list of file names, with no additional information. This option may be used if the server returns the file listing in a format that is not recognized by the control.

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 **GetFileList** method returns a list of files in the specified directory, copying the data to a string buffer. Unlike the **ReadDirectory** method that parses a directory listing, this method returns the unparsed file list data. The actual format of the data that is returned depends on the operating system and how the server implements file listings. For example, UNIX servers typically return the output from the **/bin/ls** command.

Some servers may not support file listings for any directory other than the current working directory. If an error is returned when specifying a directory name, try changing the current working directory

using the **ChangeDirectory** method and then call this method again, an empty string as the ***RemotePath*** parameter.

This method can be particularly useful when the client is connected to a server that returns file listings in a format that is not recognized by the control. The application can retrieve the unparsed file listing from the server and parse the contents. Note that if you specify the **ftpListNameOnly** option, the data will only contain a list of file names and there will be no way for the application to know if they represent a regular file or a subdirectory.

This method is supported for both FTP and SFTP (SSH) connections, however the format of the data may differ depending on which protocol is used. Most UNIX based FTP servers will not list files and subdirectories that begin with a period, however most SFTP servers will return a list of all files, even those that begin with a period.

This method will cause the current thread to block until the file listing completes, a timeout occurs or the operation is canceled.

See Also

ChangeDirectory Method, OpenDirectory Method, ReadDirectory Method

GetFilePermissions Method

Return the access permissions for a file on the server.

Syntax

object.**GetFilePermissions**(*RemoteFile*, *FilePerms*)

Parameters

RemoteFile

A string that specifies the name of the file that the access permissions are to be returned for. The filename cannot contain any wildcard characters.

FilePerms

A numeric variable which is set to the file permissions when the method returns. This parameter must be passed by reference. The file permissions are represented as bit flags, and may be one or more of the following values:

Value	Description
ftpPermWorldExecute	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.
ftpPermWorldWrite	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
ftpPermWorldRead	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
ftpPermGroupExecute	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermGroupWrite	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermGroupRead	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
ftpPermOwnerExecute	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermOwnerWrite	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right

	to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermOwnerRead	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
ftpPermSymbolicLink	The file is a symbolic link to another file. Symbolic links are special types of files found on UNIX based systems which are similar to Windows shortcuts.

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 **GetFilePermissions** method returns information about the access permissions for a specific file on the server. This method uses the STAT command to retrieve information about the specified file. If the server does not support the use of this command, an error will be returned. You can use the **Features** property to determine what features are available and/or enabled on the server.

Note that on some systems, the STAT command will not return information on files that contain spaces or tabs in the filename. In this case, the method will fail.

Example

The following example demonstrates how to retrieve the access permissions for a file and then test to see if the file can be read by the owner of that file:

```
nError = FileTransfer1.GetFilePermissions(strFileName, nFilePerms)
If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString, vbExclamation
    Exit Sub
End If

If (nFilePerms And ftpPermOwnerRead) <> 0 Then
    MsgBox "The file " & strFileName & " can be read by the owner"
End If
```

See Also

Features Property, SetFilePermissions Method

GetFileSize Method

Returns the size of the specified file on the server.

Syntax

object.**GetFileSize**(*RemoteFile*, *FileSize*)

Parameters

RemoteFile

A string that specifies the name of the file on the server. The filename cannot contain any wildcard characters and must follow the naming conventions of the operating system the server is hosted on.

FileSize

A numeric variable which will be set to the size of the file on the server. Note that if the variable is not large enough to contain the file size, an overflow error will occur. This parameter must be passed by reference.

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 **GetFileSize** method uses the SIZE command to determine the length of the specified file. Not all servers implement this command, in which case the method will fail. You can use the **Features** property to determine what features are available and/or enabled on the server.

Note that if the file on the server is a text file, it is possible that the value returned by this method will not match the size of the file when it is downloaded to the local system. This is because different operating systems use different sequences of characters to mark the end of a line of text, and when a file is transferred in text mode, the end of line character sequence is automatically converted to a carriage return-linefeed, which is the convention used by the Windows platform.

Example

The following example demonstrates how to retrieve the size a file on the server:

```
Dim nFileSize As Long

nError = FileTransfer1.GetFileSize(strFileName, nFileSize)
If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString, vbExclamation
    Exit Sub
End If

MsgBox "The size of " & strFileName & " is " & nFileSize " bytes"
```

See Also

Features Property, GetFileStatus Method, GetFileTime Method

GetFileStatus Method

Return status information about a specific file on an FTP server.

Syntax

object.**GetFileStatus**(*FileName*, [*FileLength*], [*FileDate*], [*FileOwner*], [*FileGroup*], [*FilePerms*], [*IsDirectory*])

Parameters

FileName

A string value that specifies the name of the file that status information will be returned for.

FileLength

An integer value that will specify the size of the file on the server when the method returns. This parameter must be passed by reference. Note that if this is a text file, the file size may be different on the server than it is on the local system. This is because different operating systems use different conventions that indicate the end of a line and/or the end of the file. On Windows platforms, directories have a file size of zero bytes.

FileDate

A string value that will specify the date and time the file was created or last modified on the server. This parameter must be passed by reference. The date format that is returned is expressed in local time (in other words, the timezone of the server is not taken into account) and depends on both the local host settings via the Control Panel and the format of the date and time information returned by the server

FileOwner

A string value that will specify the owner of the file on the server. This parameter must be passed by reference. On some platforms, this information may not be available for security reasons if an anonymous login session was specified

FileGroup

A string value that will specify the group that the file owner belongs to. This parameter must be passed by reference. On some platforms, this information may not be available for security reasons if an anonymous login session was specified

FilePerms

An integer value that will specify the permissions assigned to the file. This parameter must be passed by reference. This value is actually a combination of bits that specify the individual permissions for the file owner, group and world (all other users). For those familiar with UNIX, the file permissions are the same as those used by the **chmod** command. The permissions are as follows:

Value	Description
ftpPermWorldExecute	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.
ftpPermWorldWrite	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.

ftpPermWorldRead	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
ftpPermGroupExecute	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermGroupWrite	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermGroupRead	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
ftpPermOwnerExecute	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermOwnerWrite	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermOwnerRead	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
ftpPermSymbolicLink	The file is a symbolic link to another file. Symbolic links are special types of files found on UNIX based systems which are similar to Windows shortcuts.

IsDirectory

A boolean value that will specify if the file is a directory or a regular file. This parameter must be passed by reference.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **GetFileStatus** method returns information about the specified file. The filename must be specified using the server file naming conventions, and cannot include wildcard characters. The primary difference between using this method and using the **OpenDirectory** and **ReadDirectory** methods to obtain file information is that the file status information is returned on the command channel. This method cannot be used while a file transfer is in progress or while a file listing is being returned by the server. All output arguments are optional.

Note that this method requires that the server return file status information in response to the STAT

command. Some servers, for example on VMS platforms, do not provide this information. On some systems, the STAT command will not return information on files that contain spaces or tabs (whitespace) in the filename. In this case, the method will set the specified arguments to empty strings and zero values.

See Also

CloseDirectory Method, OpenDirectory Method, ReadDirectory Method

GetFileTime Method

Returns the modification date and time for specified file on the server.

Syntax

object.**GetFileTime**(*RemoteFile*, *FileDate*)

Parameters

RemoteFile

A string that specifies the name of the file on the server. The filename cannot contain any wildcard characters and must follow the naming conventions of the operating system the server is hosted on.

FileDate

A variable that will be set to the date and time that the file was last modified. The variable's data type may either be a Variant, String or Date.

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 **GetFileTime** method uses the MTDM command to determine the modification time for the file. If the server does not support this command, the function will attempt to use the STAT command to determine the file modification time. You can use the **Features** property to determine what features are available and/or enabled on the server.

The **Localize** property will determine if the returned file time is adjusted for the local timezone.

Example

The following example demonstrates how to retrieve the size a file on the server:

```
Dim dateFileTime As Date

nError = FileTransfer1.GetFileTime(strFileName, dateFileTime)
If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString, vbExclamation
    Exit Sub
End If

MsgBox strFileName & " was modified on " & dateFileTime
```

See Also

Features Property, Localize Property, GetFileStatus Method, GetFileSize Method, SetFileTime Method

GetFileType Method

Returns the file type for a file on the local system.

Syntax

object.**GetFileType**(*FileName*, *ScanFile*)

Parameters

FileName

A string which specifies the name of a file on the local system. The file does not need to exist if the *ScanFile* parameter is **False**. If an existing file is specified, it cannot be the name of a device or a directory, otherwise the method will fail.

ScanFile

An optional Boolean value which specifies if the contents of the file should be scanned. A value of **False** indicates that only the file extension should be used to determine the file type, while a value of **True** specifies the contents of the file should be examined if the file type cannot be determined based on its extension. If this parameter is omitted, the default value is **False**.

Return Value

An integer value of zero or greater which identifies the file type using the same values as the **FileType** property. If the method fails, it will return -1 indicating an error condition. The value of the **LastError** property can be used to determine the cause of the failure.

Remarks

This method is used to determine the file transfer type to be used when uploading or downloading files. This method is used internally when **fileTypeAuto** is specified as the default file type. The return value may be one of the following:

Value	Description
fileTypeASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings. The constant **fileTypeText** is an alias for this value.
fileTypeEBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.
fileTypeImage	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executable programs. If the type of file cannot be automatically determined, it will always be considered a binary file. If this file type is specified when uploading or downloading text files, the native end-of-line character sequences will be preserved. The constant **fileTypeBinary** is an alias for this value.

If the file extension or contents are not recognized, the default file transfer type for the client session

will be returned. This will usually be **fileTypeImage**, however this can be changed by calling the **AddFileType** method. The file type for the current client session can be explicitly set using the **FileType** property.

If the ***ScanFile*** parameter is True, the local file will be opened in a shared reading mode and up to 4,096 bytes will be examined to determine if it contains binary data. If the file is currently locked or has been opened exclusively by another process, the file type associated with the file extension will be returned instead. Text files which contain UTF-16 text will always return a file type of **fileTypeImage** because they can contain non-ASCII characters and/or embedded null characters.

If the ***ScanFile*** parameter is True and the file type cannot be determined based on the file name extension, the file specified by ***FileName*** must exist and be a regular file. If the file does not exist, an error will be returned and the last error code will be set to **stErrorFileNotFound**. If the ***ScanFile*** parameter is False, no errors will be returned if the file does not exist, the function will only check the file name extension to determine the file type. When downloading a file, the ***ScanFile*** parameter should normally be zero because the local file may not exist yet.

See Also

FileType Property, AddFileType Method, GetFile Method, PutFile Method

GetMultipleFiles Method

Copy multiple files from the server to the local system via FTP.

Syntax

object.**GetMultipleFiles**(*LocalDir*, *RemoteDir*, *FileMask*)

Parameters

LocalDir

A string value that specifies the name of the directory on the local system where the files will be stored. If a file by the same name already exists in the directory, it will be overwritten.

RemoteDir

A string value that specifies the name of the directory on the server where the files will be copied from. You must have permission to read the contents of the directory.

FileMask

A string value that specifies which files will be copied from the server. Typically this is a wildcard pattern, such as "*.exe", which would specify all executable programs on a Windows system.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **GetMultipleFiles** method copies multiple files from the server to the local system. If the local file already exists, it is overwritten.

Example

```
nError = FileTransfer1.GetMultipleFiles(strLocalDir, strServerDirectory, "*.wav")
```

See Also

GetFile Method, PutMultipleFiles Method

GetText Method

Download a text file from the server and store it in string.

Syntax

object.**GetText**(*RemoteFile*, *Buffer*)

Parameters

RemoteFile

A string that specifies the name of a file on the server that will be downloaded. The file pathing and name conventions must be that of the server.

Buffer

This parameter is passed by reference and specifies the string buffer which will contain the text returned by the server. This parameter must be a String or Variant type which will reference a string when the method returns. This method will not accept a byte array as an argument.

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 **GetText** method is used to download the contents of a text file and store it in a String variable. This method should only be used with text files which are known to be textual. For example, it is safe to use this method when downloading an HTML or XML document, but should not be used to download executable or compressed files (such as Microsoft Word documents or Excel spreadsheets) . Always use the **GetData** method if you wish to retrieve binary data and store it in a byte array.

The text document returned by the server is automatically converted to Unicode using the code page specified by the **CodePage** property. Most text files today will use either ASCII or UTF-8 encoding, however some documents may contain text specific to the locale they were created in. Because ASCII is a subset of UTF-8, it is safe to specify UTF-8 encoding for ASCII text documents. If you specify an incorrect code page, this can result in a conversion error.

This method will always attempt to normalize the end-of-line character sequence to use a carriage-return and linefeed (CRLF) pair. This can potentially result in a discrepancy between the size of a text file on the server and the actual length of the string buffer.

This method will always use an ASCII file transfer mode, regardless of the value of the **FileType** property. If the remote file contains binary data, the string buffer may be empty or contain unprintable characters as the result of attempting to convert the data to Unicode.

See Also

CodePage Property, FileType Property, GetData Method, PutData Method, PutText Method, OnProgress Event

Initialize Method

Initialize the component and load the networking library.

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 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 fileTransfer = CreateObject("SocketTools.FileTransfer.11")

nError = fileTransfer.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

Uninitialize Method

MakeDirectory Method

Create a new directory on the remote FTP host.

Syntax

object.**MakeDirectory**(*NewDirectory*)

Parameters

NewDirectory

A string value that specifies the name of the directory to create on the server.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **MakeDirectory** method creates a new directory on the server. The user must have the appropriate permissions to create a directory or an error will occur.

Example

```
Private Sub cmdMakeDir_Click()
    Dim nError As Long

    nError = FileTransfer1.MakeDirectory(Trim(txtNewDirectory.Text))
    If nError > 0 Then
        MsgBox "MakeDirectory error: " & nError
    End If
End Sub
```

See Also

ChangeDirectory Method, RemoveDirectory Method

OpenDirectory Method

Open the specified directory on an FTP server.

Syntax

object.**OpenDirectory**([*DirName*], [*ParseList*])

Parameters

DirName

An optional string value that specifies the directory on the server. If it is an empty string, it designates the current remote directory. It may also be a file mask that includes wildcards if supported by the server. If this argument is omitted, then the current working directory will be opened.

ParseList

An optional boolean value that determines if the directory listing will be parsed as it is read. The default value is true. If the ***ParseList*** parameter is false, then each entire un-parsed line of a directory listing will be returned in the ***FileName*** parameter of the **ReadDirectory** method, and all other output parameters will be empty.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **OpenDirectory** method opens the specified directory on the server so that the list of files in that directory may be read.

Note that you must call the **CloseDirectory** method after the list of files has been returned by the server. Failure to do so will result in an error when you attempt to transfer a file because the data channel to the server has been left open. For this same reason, you cannot call the **GetFile** or **PutFile** methods while reading the contents of a directory on the server.

Example

```
' Request a parsed directory listing.
nError = FileTransfer1.OpenDirectory(strDirName)

' Request a parsed directory listing of the current directory
nError = FileTransfer1.OpenDirectory()

' Request an un-parsed listing of files in the current directory
nError = FileTransfer1.OpenDirectory("", False)

' Request a parsed listing of files in the current directory
' with file extension .exe
nError = FileTransfer1.OpenDirectory("*.exe")
```

See Also

CloseDirectory Method, ReadDirectory Method

PostFile Method

Post the contents of the specified file to a script executed on the server.

Syntax

object.**PostFile**(*LocalFile*, [*Resource*], [*FieldName*], [*Options*])

Parameters

LocalFile

A string that specifies the file on the local system that will be transferred to the server. The file pathing and name conventions must be that of the local host.

Resource

A string that specifies the resource that the data will be posted to on the server. Typically this is the name of an executable script. This is an optional argument; if it is omitted, the value of the **Resource** property will be used. It is also permissible to specify a complete URL and the file will be uploaded to that location.

FieldName

An optional string argument that specifies the form field name that the script expects. If this argument is omitted or is an empty string, a default field name of "File1" is used. This is an optional argument; if it is omitted, the value of the **Resource** property will be used. It is also permissible to specify a complete URL and the file will be downloaded from that location.

Options

An optional numeric value which specifies one or more options. This argument is reserved for future use and the application should either omit this argument, or specify a value of zero.

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 **PostFile** method posts the contents of a file to a script that is executed on the server. This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

This method is similar to the **PutFile** method in that it can be used to upload the contents of a local file to a server. However, instead of using the PUT command, the POST command is used to send the file data to a script that is executed on the server. This method has the advantage of not requiring any special configuration settings on the server, however it does require that the script be able to process multipart/form-data as defined in RFC 2388.

To support uploading files from a form on a webpage, the FILE input type is used along with the action that specifies the script that will accept the file data and process it. For example, the HTML code could look like this:

```
<form action="/cgi-bin/upload.cgi" method="post" enctype="multipart/form-
data">
<input type="file" name="datafile" size="20">
<input type="submit">
</form>
```

In this example, the script /cgi-bin/upload.cgi is responsible for processing the file data that is posted by the client, and the form field name "datafile" is used. The user can select a file, and when the Submit button is clicked, the file data is posted to the script. To simulate this using the **PostFile** method, the ***LocalFile*** argument should be set to the name of the local file that will be posted to the server. The ***Resource*** argument should be the name of the script, in this case "/cgi-bin/upload.cgi". The ***FieldName*** argument should be specified as the string "datafile" to match the name of the field used by the form.

Note that the **PostFile** function always submits the file contents as multipart/form-data with the content type set to application/octet-stream. The script that accepts the posted data must be able to parse the multipart header block and correctly process 8-bit data. If the script assumes that the data will be posted using a specific encoding type such as base64, then the file data may not be accepted or may be corrupted by the script.

See Also

Resource Property, URL Property, GetFile Method, PutFile Method, PutMultipleFiles Method, OnProgress Event

PutData Method

Transfers data from a local buffer and stores it in a file on the server.

Syntax

object.**PutData**(*RemoteFile*, *Buffer*, [*Length*], [*Reserved*])

Parameters

RemoteFile

A string that specifies the file on the server that will contain the data being transferred. If the file already exists, it will be overwritten. The file pathing and name conventions must be that of the server.

Buffer

This parameter specifies the local buffer that the data will be copied from. 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 file on the server is a text file. If the remote file should contain binary data, it is recommended that a Byte array variable be specified as the argument to this method.

Length

An optional integer argument that specifies the amount of data to be copied from the buffer. If this argument is omitted, the entire contents of the buffer is transferred to the server.

Reserved

An argument reserved for future expansion. This argument should always be omitted or specified as a numeric value of zero.

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 **PutData** method transfers data from a local buffer and stores it on a file on the server. This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

Priority Property, GetData Method, GetFile Method, PutFile Method, OnProgress Event

PutFile Method

Copy a file from the local system to the server.

Syntax

object.**PutFile**(*LocalFile*, [*RemoteFile*], [*Offset*])

Parameters

LocalFile

A string that specifies the name of the file that will be uploaded from the local system. The file pathing and name conventions must be that of the local host. You must have permission to open this file for reading.

RemoteFile

A string that specifies specifies the name of the file to create on the server. You must have permission to create or overwrite the file. This is an optional argument; if it is omitted, the value of the **Resource** property will be used. It is also permissible to specify a complete URL and the file will be uploaded to that location.

Offset

An optional integer argument that specifies the byte offset where the file transfer will begin. This argument is only valid for FTP servers, and is used to resume interrupted transfers.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **PutFile** method copies an existing file from the local system to the server. If the local file already exists, it is overwritten.

Note that not all servers honor the request to restart a file transfer. Notably, a Windows IIS server will return an error if a non-zero restart offset is specified. It is not recommended that you restart text file transfers since differences between end-of-line characters can result in byte offset differences between the local and server system.

If the **AppendFile** property is True, then the file contents will be appended to the file specified on the server, if it exists. Otherwise, the remote file will be overwritten.

Note that **PutFile** for HTTP requires that the server support the PUT command. This typically requires that you specify version 1.1 of the protocol and authenticate the client session with a user name and password.

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.

Example

```
nError = FileTransfer1.PutFile(strLocalFile, strRemoteFile)
If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

AppendFile Property, URL Property, GetData Method, GetFile Method, PostFile Method, PutData Method, PutMultipleFiles Method

---

PutMultipleFiles Method

Copy multiple files from the local system to the server via FTP.

Syntax

object.**PutMultipleFiles**(*LocalDir*, *RemoteDir*, [*FileMask*])

Parameters

LocalDir

A string value that specifies the name of the directory on the local system where the files are stored. You must have permission to read the contents of the directory.

RemoteDir

A string value that specifies the name of the directory on the server where the files will be copied to. If a file by the same name already exists in this directory, it will be overwritten.

FileMask

An optional string value specifies which files will be copied from the server. Typically this is a wildcard pattern, such as "*.exe", which would specify all executable programs on a Windows system.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **PutMultipleFiles** method copies multiple files from the local system to the server. If the remote file already exists, it is overwritten.

Example

```
nError = FileTransfer1.PutMultipleFiles(strLocalDir, strServerDirectory, "*.wav")
```

See Also

PostFile Method, PutFile Method, GetMultipleFiles Method

PutText Method

Upload the contents of a string buffer and store it in a text file on the server.

Syntax

object.**PutText**(*RemoteFile*, *Buffer*)

Parameters

RemoteFile

A string that specifies the name of a file on the server that will be downloaded. The file pathing and name conventions must be that of the server.

Buffer

A string which contains the text to be stored on the server. This method will not accept a Byte array as an argument.

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 **PutText** method is used to upload the contents of a string and store it as a text file on the server. Although a String variable may contain binary data, this method should only be used with strings which contain printable text. Always use the **PutData** method if you wish to upload binary data, using a Byte array instead of a String variable.

The text uploaded to the server is automatically converted from Unicode using the code page specified by the **CodePage** property. By default, text will be automatically converted to use UTF-8 encoding, however you can change this if you prefer to store the file using a different localized encoding. In most cases it is recommended you use UTF-8 to ensure the broadest compatibility with other applications.

This method will always attempt to normalize the end-of-line character sequence to match what is used on the server. This can potentially result in a discrepancy between the size of a text file on the server and the actual length of the string buffer. For example, Windows uses a carriage return and linefeed pair (CRLF) to indicate the end of a line of text. If you are storing the text in a file on a UNIX system, it will be changed to use only a linefeed (LF) to indicate the end of a line.

This method will always use an ASCII file transfer mode, regardless of the value of the **FileType** property. If the string buffer contains binary data, the resulting file may be empty or contain unprintable characters as the result of the Unicode text conversion.

See Also

CodePage Property, FileType Property, GetData Method, GetText Method, PutData Method, OnProgress Event

ReadDirectory Method

Read a directory entry from an FTP server.

Syntax

object.**ReadDirectory**(*FileName*, [*FileLength*], [*FileDate*], [*FileOwner*], [*FileGroup*], [*FilePerms*], [*IsDirectory*])

Parameters

FileName

A string value that specifies the name of the file that status information will be returned for.

FileLength

An integer value that will specify the size of the file on the server when the method returns. This parameter must be passed by reference. Note that if this is a text file, the file size may be different on the server than it is on the local system. This is because different operating systems use different conventions that indicate the end of a line and/or the end of the file. On Windows platforms, directories have a file size of zero bytes.

FileDate

A string value that will specify the date and time the file was created or last modified on the server. This parameter must be passed by reference. The date format that is returned is expressed in local time (in other words, the timezone of the server is not taken into account) and depends on both the local host settings via the Control Panel and the format of the date and time information returned by the server

FileOwner

A string value that will specify the owner of the file on the server. This parameter must be passed by reference. On some platforms, this information may not be available for security reasons if an anonymous login session was specified

FileGroup

A string value that will specify the group that the file owner belongs to. This parameter must be passed by reference. On some platforms, this information may not be available for security reasons if an anonymous login session was specified

FilePerms

An integer value that will specify the permissions assigned to the file. This parameter must be passed by reference. This value is actually a combination of bits that specify the individual permissions for the file owner, group and world (all other users). For those familiar with UNIX, the file permissions are the same as those used by the **chmod** command. The permissions are as follows:

Value	Description
ftpPermWorldExecute	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.
ftpPermWorldWrite	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.

ftpPermWorldRead	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
ftpPermGroupExecute	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermGroupWrite	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermGroupRead	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
ftpPermOwnerExecute	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermOwnerWrite	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermOwnerRead	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
ftpPermSymbolicLink	The file is a symbolic link to another file. Symbolic links are special types of files found on UNIX based systems which are similar to Windows shortcuts.

IsDirectory

A boolean value that will specify if the file is a directory or a regular file. This parameter must be passed by reference.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ReadDirectory** method reads the next entry from the directory listing. This method can only be used after the **OpenDirectory** method has been called to begin the transfer of file information to the client. All parameters of **ReadDirectory** are output parameters. The first parameter, ***FileName***, is required, and all other are optional. If the ***ParseList*** parameter of the **OpenDirectory** method was missing, or was explicitly set to True, then the output parameters that are present will have the meanings described below. If the **ParseList** parameter of the **OpenDirectory** method was False, then the ***FileName*** parameter will contain all file information, exactly as provided by the server, without

interpretation.

For the proprietary Sterling directory formats, the "status code" is returned in the ***FilePerms*** argument. This value is a combination of bits. Bits 0-25 correspond to letters of the alphabet, most of which have distinct meanings in the Sterling formats.

Letter code	Bit position	Hexadecimal value
A	1h	
B	2h	
C	4h	
n-th letter of alphabet	n-1	2 to the (n-1) power
Z	2000000h	

For the proprietary Sterling directory formats, bits 26-31 represent the transfer protocol associated with the file:

Protocol	Bit position	Hexadecimal value	Value
TCP	4000000h	ftpSterlingStatusTcp	
FTP	8000000h	ftpSterlingStatusFtp	
BSC	10000000h	ftpSterlingStatusBsc	
ASC	20000000h	ftpSterlingStatusAsc	
FTS	40000000h	ftpSterlingStatusFts	
other	80000000h	ftpSterlingStatusOther	

Certain error codes should be treated as normal terminations of a directory listing. If the **OpenDirectory** method was called with the ***ParseList*** parameter specified as True, or if the ***ParseList*** parameter is omitted, then **ReadDirectory** will return the error **fileErrorEndOfDirectory** when the end of the directory listing is reached. If the **OpenDirectory** method was called with the ***ParseList*** parameter specified as False, then the **ReadDirectory** method will return the error **fileErrorEndOfData** when there are no more filenames to return.

You must call the **CloseDirectory** method after the list of files has been returned by the server. Failure to do so will result in an error when you attempt to transfer a file because the data channel to the server has been left open. For this same reason, you cannot call the **GetFile** or **PutFile** methods while reading the contents of a directory on the server.

Example

Example 1:

```
'
' Display full file listing for specified directory
'
nError = FileTransfer1.OpenDirectory(strDirName)
If nError > 0 Then
    MsgBox "OpenDirectory error: " & nError
    Exit Sub
End If

Do
    nError = FileTransfer1.ReadDirectory(strFileName, _
```

```
                                                dwFileLength, strFileDate, _
                                                strFileOwner, strFileGroup, _
                                                dwFilePerms, bIsDirectory)
        If nError > 0 Then
            If nError <> fileErrorEndOfDirectory Then
                MsgBox "ReadDirectory error: " & nError
            End If
            Exit Do
        End If

        '
        ' See GetFileStatus help topic for FilePerms function
        '
        strPerms = FilePerms(dwFilePerms, bIsDirectory)
        txtFileStatus.Text = txtFileStatus.Text & _
            strFileName & " " & dwFileLength & " " & strFileDate & " " & _
            strFileOwner & " " & strFileGroup & " " & strPerms & vbCrLf
    Loop
    FileTransfer1.CloseDirectory
```

Example 2:

```
    '
    ' Display file names and dates for specified directory
    '
    nError = FileTransfer1.OpenDirectory(strDirName)
    If nError > 0 Then
        MsgBox "OpenDirectory error: " & nError
        Exit Sub
    End If

    Do
        nError = FileTransfer1.ReadDirectory(strFileName, , strFileDate)
        If nError > 0 Then
            If nError <> fileErrorEndOfDirectory Then
                MsgBox "ReadDirectory error: " & nError
            End If
            Exit Do
        End If
        txtFileStatus.Text = txtFileStatus.Text & _
            strFileName & " " &  strFileDate & vbCrLf
    Loop
    FileTransfer1.CloseDirectory
```

Example 3:

```
    '
    ' Display unparsed file listing for specified directory
    '
    nError = FileTransfer1.OpenDirectory(strDirName, False)
    If nError > 0 Then
        MsgBox "OpenDirectory error: " & nError
        Exit Sub
    End If

    Do
        nError = FileTransfer1.ReadDirectory(strFileDescription)
        If nError > 0 Then
            If nError <> fileErrorEndOfData Then
                MsgBox "ReadDirectory error: " & nError
            End If
```

```
            Exit Do
        End If
        txtFileStatus.Text = txtFileStatus.Text & strFileDescription & vbCrLf
    Loop
    FileTransfer1.CloseDirectory
```

See Also

GetFileStatus Method, CloseDirectory Method, OpenDirectory Method

RemoveDirectory Method

Remove a directory on the remote FTP server.

Syntax

object.**RemoveDirectory**(*DirectoryName*)

Parameters

DirectoryName

A string value which specifies the name of the directory to remove on the server.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **RemoveDirectory** method removes an existing directory on the server. You must have the appropriate permission to remove the directory, or an error will occur. Note that most systems will not allow you to remove a directory if it contains files.

Example

```
Private Sub cmdRmdir_Click()
    Dim nError As Long

    nError = FileTransfer1.RemoveDirectory(Trim(txtDirectory.Text))
    If nError > 0 Then
        MsgBox "RemoveDirectory error: " & nError
    End If
End Sub
```

See Also

ChangeDirectory Method, MakeDirectory Method

RenameFile Method

Change the name of an existing file on the FTP server.

Syntax

object.**RenameFile**(*OldFileName*, *NewFileName*)

Parameters

OldFileName

A string value that specifies the current name of the file on the server.

NewFileName

A string value that specifies the name that the existing file will be changed to.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **RenameFile** method renames an existing file on the server to the new name. You must have permission to change the file name or an error will occur.

Example

```
nError = FileTransfer1.RenameFile(strOldFileName, strNewFileName)
If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString
End If
```

See Also

DeleteFile 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, open network connections will be closed and any handles allocated by the control will be released.

The **Reset** and **Uninitialize** methods will abort all active background transfers and wait for those tasks to complete before returning to the caller. It is recommended that your application explicitly wait for background transfers to complete or abort them using this method before allowing the program to terminate. This will ensure that your program can perform any necessary cleanup operations. If there are active background tasks running at the time that the control instance is destroyed, it can force the control to stop those worker threads immediately without waiting for them to terminate gracefully.

See Also

Cancel Method

SetFilePermissions Method

Change the access permissions for a file on the server.

Syntax

object.**SetFilePermissions**(*RemoteFile*, *FilePerms*)

Parameters

RemoteFile

A string that specifies the name of the file that the access permissions are to be returned for. The filename cannot contain any wildcard characters.

FilePerms

A numeric value which specifies the new permissions for the file. The file permissions are represented as bit flags, and may be one or more of the following values:

Value	Description
ftpPermWorldExecute	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.
ftpPermWorldWrite	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
ftpPermWorldRead	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
ftpPermGroupExecute	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermGroupWrite	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermGroupRead	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
ftpPermOwnerExecute	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermOwnerWrite	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this

	grants the user the right to create and delete files.
ftpPermOwnerRead	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.

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 **SetFilePermissions** method uses the SITE CHMOD command to set the permissions for the file. This command is typically only supported on servers that are hosted on UNIX based systems. If the command is not supported, an error will be returned. You can use the **Features** property to determine what features are available and/or enabled on the server.

Users who are familiar with the UNIX operating system will recognize the **chmod** command used to change the file permissions. However, it should be noted that the numeric value used as an argument to the command is in octal, not decimal. For example, issuing the command **chmod 644 filename.txt** on a UNIX based system will make the file readable and writable by the owner, and readable by other users in the owner's group as well as all other users. The value 644 is an octal value, which is equivalent to the decimal value 420. If you were to mistakenly specify 644 as the value for the ***Permissions*** argument, rather than the decimal value of 420, the permissions on the file would be incorrect. It is strongly recommended that you use the pre-defined constants to prevent this sort of error.

Visual Basic allows you to specify an integer value in octal by prefixing it with &O. For example, &O644 could be used as the file permissions value. C and C++ consider any integer with a preceding 0 to be an octal number, so 0644 would be a valid permissions value. Consult the technical reference for your programming language if you are unsure if it supports expressing integer constants in octal.

Example

The following example demonstrates how to change the permissions so that only the owner can read and write to the file:

```
nFilePerms = ftpPermOwnerRead Or ftpPermOwnerWrite
nError = FileTransfer1.SetFilePermissions(strFileName, nFilePerms)
If nError > 0 Then
    MsgBox FileTransfer1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

Features Property, GetFilePermissions Method

SetFileTime Method

Changes the modification date and time for a file on the server.

Syntax

object.**SetFileTime**(*RemoteFile*, *FileTime*)

Parameters

RemoteFile

A string that specifies the name of the file on the server. The filename cannot contain any wildcard characters and must follow the naming conventions of the operating system the server is hosted on.

FileTime

A string that specifies the new date and time for the file. The date must be in a format recognized by the local system, otherwise an error will occur. The date and time value must also be specified in UTC (Coordinated Universal Time), not local time.

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 **SetFileTime** method changes the modification date and time for the specified file on the server. When connected to an FTP server, this method uses the MTDM command to change the modification time for the file. If the server does not support this command, the method will return an error. Note that some servers only support the MDTM command to return, but not change, the file modification time.

See Also

Localize Property, GetFileStatus Method, GetFileSize Method, GetFileTime Method

TaskAbort Method

Abort the specified asynchronous task.

Syntax

object.**TaskAbort** ([*TaskId*], [*Milliseconds*])

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

Milliseconds

An optional integer value that specifies the number of milliseconds to wait for the background task to abort.

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 **TaskAbort** method signals the background worker thread associated with the task ID to abort the current operation and terminate as soon as possible. If the ***TaskId*** parameter is omitted, this method will abort all active background file transfers, otherwise it will only abort the specified task. If the ***Milliseconds*** parameter is omitted or has a value of zero, the method returns immediately after the background thread has been signaled. If the ***Milliseconds*** parameter is non-zero, the method will wait that amount of time for the background thread to terminate.

The **Reset** and **Uninitialize** methods will abort all active background transfers and wait for those tasks to complete before returning to the caller. It is recommended that your application explicitly wait for background transfers to complete or abort them using this method before allowing the program to terminate. This will ensure that your program can perform any necessary cleanup operations. If there are active background tasks running at the time that the control instance is destroyed, it can force the control to stop those worker threads immediately without waiting for them to terminate gracefully.

See Also

TaskCount Property, TaskList Property, TaskDone Method, TaskWait Method

TaskDone Method

Determine if an asynchronous task has completed.

Syntax

object.**TaskDone** ([*TaskId*])

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

Return Value

A Boolean value that specifies if the task has completed. A return value of **True** specifies that the background task has completed. A return value of **False** specifies that the background task is active.

Remarks

The **TaskDone** method is used to determine if the specified asynchronous task has completed. If the *TaskId* parameter is omitted, the method will check the status of the last background task that was started.

If you use this method to poll the status of a background task from within the main UI thread, you must ensure that Windows messages are processed so that the application remains responsive to the end-user. To check if a background transfer has completed, it is recommended that you use a timer to periodically call this method rather than calling it repeatedly within a loop.

To determine if the task completed successfully, the **TaskWait** method will provide the last error code associated with the task. Note that if this method returns **True**, it is guaranteed that calling **TaskWait** using the same task ID will return the error code to the caller immediately without causing the application to block.

See Also

TaskCount Property, TaskId Property, TaskList Property, TaskAbort Method, TaskWait Method

TaskResume Method

Resume execution of an asynchronous task.

Syntax

object.**TaskResume** (*TaskId*)

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

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 **TaskResume** method resumes execution of the background worker thread that was previously suspended using the **TaskSuspend** method. If the ***TaskId*** parameter is omitted, the method will resume execution of the last background task that was started.

See Also

TaskId Property, TaskSuspend Method, TaskWait Method

TaskSuspend Method

Suspend execution of an asynchronous task.

Syntax

object.**TaskSuspend** (*TaskId*)

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

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 **TaskSuspend** method will suspend execution of the background worker thread associated with the task. If the ***TaskId*** parameter is omitted, the method will suspend the last background task that was started.

Once the task has been suspended, it will no longer be scheduled for execution, however the client session will remain active and the task may be resumed using the **TaskResume** method. Note that if a task is suspended for a long period of time, the background operation may fail because it has exceeded the timeout period imposed by the server.

See Also

TaskId Property, TaskResume Method, TaskWait Method

TaskWait Method

Wait for an asynchronous task to complete.

Syntax

object.**TaskWait** ([*TaskId*], [*Milliseconds*], [*TimeElapsed*], [*TaskError*])

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

Milliseconds

An optional integer value that specifies the number of milliseconds to wait for the background task to complete.

TimeElapsed

An optional integer value passed by reference that will contain the elapsed time for the task in milliseconds when the method returns. If this information is not required, this parameter may be omitted. This parameter is ignored if the ***TaskId*** parameter is omitted.

TaskError

An optional integer value passed by reference that will contain the last error code for the task when the method returns. If this information is not required, this parameter may be omitted. This parameter is ignored if the ***TaskId*** parameter is omitted.

Return Value

A Boolean value that specifies if the task has completed. A return value of **True** specifies that the background task has completed. A return value of **False** specifies that the background task is active.

Remarks

The **TaskWait** method waits for the specified task to complete. If the ***TaskId*** parameter is omitted, this method will wait for all active tasks to complete. If a task ID is specified and the ***Milliseconds*** parameter is non-zero, this method will cause the current working thread to block until the task completes or the amount of time exceeds the number of milliseconds specified by the caller. If the ***Milliseconds*** parameter is zero, then this function will poll the status of the task and return immediately to the caller. If the ***Milliseconds*** parameter is omitted, then the method will wait an infinite period of time for the task to complete.

If the specified task has already completed at the time this method is called, the method will return immediately without causing the current thread to block. If the ***TimeElapsed*** parameter has been specified, it will contain the number of milliseconds that it took for the task to complete. If the ***TaskError*** parameter has been specified, it will contain the last error code value that was set by the worker thread before it terminated. If the ***TaskError*** value is zero, that means that the background task was successful and no error occurred. A non-zero value will indicate that the background task has failed.

You should not call this method from the main UI thread with a long timeout period to wait for a background task to complete. Windows messages will not be processed while this method is blocked waiting for the background task to complete, and this can cause your application to appear non-responsive to the end-user. If you have a GUI application and you need to determine if a background task has finished, create a timer to periodically call the **TaskDone** method. When it returns **True** (indicating that the task has completed), you can safely call **TaskWait** to obtain the elapsed time and last error code without blocking the current thread.

See Also

TaskCount Property, TaskList Property, TaskAbort Method, TaskDone 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. 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.

The **Reset** and **Uninitialize** methods will abort all active background transfers and wait for those tasks to complete before returning to the caller. It is recommended that your application explicitly wait for background transfers to complete or abort them using this method before allowing the program to terminate. This will ensure that your program can perform any necessary cleanup operations. If there are active background tasks running at the time that the control instance is destroyed, it can force the control to stop those worker threads immediately without waiting for them to terminate gracefully.

See Also

Initialize Method

VerifyFile Method

Verify that the contents of a file on the local system are the same as the specified file on the server..

Syntax

object.**VerifyFile**(*LocalFile*, *RemoteFile*, [*Options*])

Parameters

LocalFile

A string that specifies the name of the file on the local system.

RemoteFile

A string that specifies the name of the file on the server.

Options

A numeric bitmask which specifies the options that may be used when comparing the files. This argument may be any one of the following values:

Value	Description
fileVerifyDefault	File verification should use the best option available based on the available server features. If the server supports the XMD5 command, the control will calculate an MD5 has of the local file contents and compare the value with the file on the server. If the server does not support the XMD5 command, but it does support the XCRC command, the control will calculate a CRC32 checksum of the local file contents and compare the value with the file on the server. If the server does not support either the XMD5 or XCRC commands, the control will compare the size of the local and remote files.
fileVerifySize	Files are verified by comparing the number of bytes of data in the local and remote files. This is the least reliable method, and should only be used if the server does not support either the XMD5 or XCRC commands.
fileVerifyCRC32	Files are verified by calculating a CRC-32 checksum of the local file contents and comparing it with the value returned by the server in response to the XCRC command. This method should only be used if the server does not support the XMD5 command.
fileVerifyMD5	Files are verified by calculating an MD5 hash of the local file contents and comparing it with the value returned by the server in response to the XMD5 command. This is the preferred method for performing file verification.

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 **VerifyFile** method will attempt to verify that the contents of the local and remote files are identical using one of several methods, based on the features that the server supports. Preference will be given to the most reliable method available, using either an MD5 hash, a CRC-32 checksum or comparing the size of the file, in that order.

It is not recommended that you use this method with text files because of the different end-of-line conventions used by different operating systems. For example, a text file on a Windows system uses a carriage-return and linefeed pair to indicate the end of a line of text. However, on a UNIX system, a single linefeed is used to indicate the end of a line. This can cause the **VerifyFile** method to indicate the files are not identical, even though the only difference is in the end-of-line characters that are used.

See Also

Priority Property, GetData Method, GetFile Method, GetMultipleFiles Method, PutData Method, PutFile Method, OnProgress Event

File Transfer Control Events

Event	Description
OnCancel	This event is generated when an operation is canceled
OnCommand	This event is generated when the server processes a command issued by the client
OnError	This event is generated when an error occurs
OnProgress	This event is generated when retrieving or sending files
OnTaskBegin	This event is generated when a background task begins
OnTaskEnd	This event is generated when a background task completes
OnTaskRun	This event is generated while a background task is active
OnTimeout	This event is generated after an operation times out

OnCancel Event

The **OnCancel** event is generated when an operation is canceled.

Syntax

Private Sub *object*_**OnCancel**([*Index* **As Integer**])

Remarks

The **OnCancel** event is generated after an operation is canceled by calling the **Cancel** method.

Example

```
Private Sub FileTransfer1_OnCancel()
    lblFileStatus.Caption = "Operation canceled"
End Sub
```

See Also

OnError Event, Cancel Method

OnCommand Event

The **OnCommand** event is generated when the client receives a response to a command from the server.

Syntax

Private Sub *object*_**OnCommand**([*Index* **As Integer**,] *ByVal ResultCode* **As Variant**, *ByVal ResultString* **As Variant**)

Remarks

The **OnCommand** event is generated when the client receives a reply from the server after some action has been taken. The ***ResultCode*** argument contains the numeric result code returned by the server. The result codes returned from an FTP or HTTP server fall into one of the following categories:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

The ***ResultString*** argument contains the descriptive string returned by the server which describes the result. The string contents may vary depending on the type of server.

The **ResultCode** property and **ResultString** property contain the most recent server responses. Use of the **OnCommand** event will allow the application to receive intermediate responses as well.

Example

```
Private Sub FileTransfer1_OnCommand(ByVal ResultCode As Variant, _
                                    ByVal ResultString As Variant)
    Dim startpos As Integer
    Dim crlfpos As Integer

    '
    ' Command response to debug window
    '
    Debug.Print ResultCode & " " & ResultString

    '
    ' The text control needs a little help with line terminators
    ' in multi-line responses
    '
    txtResultStream.Text = txtResultStream.Text & ResultCode & " "
    startpos = 1
    Do
        crlfpos = InStr(startpos, ResultString, Chr(10))
        If crlfpos > 0 Then
```

```
            txtResultStream.Text = txtResultStream.Text & _
                Mid(ResultString, startpos, crlfpos - startpos) & vbCrLf
            startpos = crlfpos + 1
        Else
            txtResultStream.Text = txtResultStream.Text & _
                Mid(ResultString, startpos) & vbCrLf
            Exit Do
        End If
    Loop
End Sub
```

See Also

ResultCode Property, ResultString Property, Command Method

OnError Event

The **OnError** event is generated when an error occurs.

Syntax

Private Sub *object*_**OnError**([*Index* **As Integer**,] *ByVal Error* **As Variant**, *ByVal Description* **As Variant**)

Remarks

The **OnError** event is generated when an error occurs while the component is performing an operation. 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. This corresponds to the **LastErrorString** property.

Example

```
Private Sub FileTransfer1_OnError(ByVal Error As Variant, ByVal Description As
Variant)
        Debug.Print "Error " & Error & ": " & Description
End Sub
```

See Also

LastError Property, LastErrorString Property

OnProgress Event

The **OnProgress** event is generated when retrieving or sending files.

Syntax

Private Sub *object*_**OnProgress**([*Index* **As Integer**,] *ByVal FileName* **As Variant**, *ByVal FileSize* **As Variant**, *ByVal BytesCopied* **As Variant**, *ByVal Percent* **As Variant**)

Remarks

The **OnProgress** event is generated during a file transfer and can be used to update the user interface, such as displaying a progress bar during the transaction. To cancel the current file transfer, the application can call the **Cancel** method within the event handler. The following arguments are passed to the event:

The *FileName* argument is the name of the file being transferred. If the **GetMultipleFiles** or **PutMultipleFiles** methods were used, each individual file name is returned instead of the wildcard mask.

The *FileSize* argument is a long integer which specifies the size of the file in bytes that is currently being sent or received. This value may be zero if the control cannot obtain the size of the file from the server. If the total number of bytes is less than 2 GiB, the value will be a **Long** (32-bit) integer. For very large transfers, it will be a **Double** floating-point value.

The *BytesCopied* argument is a long integer which specifies the number of bytes that have been sent or received for the current file transfer. If the number of bytes copied is less than 2 GiB, the value will be a **Long** (32-bit) integer. For very large transfers, it will be a **Double** floating-point value.

The *Percent* argument is an integer which specifies the completion percentage between a value of 0 and 100. When the **GetMultipleFiles** or **PutMultipleFiles** methods are being used, this value refers to the transfer status of a single file, not a percentage of the total number of files being transferred.

Example

```
Private Sub FileTransfer1_OnProgress(ByVal FileName As Variant, _
                                     ByVal FileSize As Variant, _
                                     ByVal BytesCopied As Variant, _
                                     ByVal Percent As Variant)

    txtFileStatus.Text = txtFileStatus.Text & _
        FileName & " " & FileSize & " " & _
        BytesCopied & " " & Percent & " " & _
        FileTransfer1.TransferTime & vbCrLf

    txtFileStatus.SelStart = Len(txtResultStream.Text)
    txtFileStatus.SelLength = 0

    If FileSize > 0 Then
        ProgressBar1.Value = Percent
    End If
End Sub
```

See Also

Cancel Method

OnTaskBegin Event

The **OnTaskBegin** event occurs when a background task starts.

Syntax

Sub *object*_**OnTaskBegin** ([*Index* **As Integer**], **ByVal** *TaskId* **As Variant**)

Remarks

The **OnTaskBegin** event is generated when a background task associated with an asynchronous file transfer begins running. The arguments to this event are:

TaskId

An integer value that uniquely identifies the background task.

This event can be used in conjunction with the **OnTaskEnd** event to monitor one or more background tasks that are created to perform asynchronous file transfers. The task ID passed to this event can be used to uniquely identify the task and corresponds to the worker thread that has been created to manage the client session. The application should consider the ID to be an opaque value and never make assumptions about how an ID is assigned to a background task.

See Also

AsyncGetFile Method, AsyncPutFile Method, OnTaskEnd Event, OnTaskRun Event

OnTaskEnd Event

The **OnTaskEnd** event occurs when a background task completes.

Syntax

Sub *object*_**OnTaskEnd** ([*Index* **As Integer**], **ByVal** *TaskId* **As Variant**, **ByVal** *TimeElapsed* **As Variant**, **ByVal** *ErrorCode* **As Variant**)

Remarks

The **OnTaskEnd** event is generated when a file transfer completes and the background task has terminated. The arguments to this event are:

TaskId

An integer value that uniquely identifies the background task.

TimeElapsed

An integer value that specifies the amount of elapsed time in milliseconds.

ErrorCode

An integer value that specifies the last error code for the task.

This event can be used in conjunction with the **OnTaskBegin** event to monitor one or more background tasks that are created to perform asynchronous file transfers. The ***TimeElapsed*** parameter will specify the number of milliseconds that the background task was active. The ***ErrorCode*** parameter specifies the last error code associated with the background task. If this value is zero, that indicates that the task completed successfully. A non-zero value indicates that the task failed and the error code value identifies why the task failed.

See Also

AsyncGetFile Method, AsyncPutFile Method, OnTaskBegin Event, OnTaskRun Event

OnTaskRun Event

The **OnTaskRun** event occurs while a background task is active.

Syntax

Sub *object*_**OnTaskRun** ([*Index* **As Integer**], **ByVal** *TaskId* **As Variant**, **ByVal** *TimeElapsed* **As Variant**, **ByVal** *Completed* **As Variant**)

Remarks

The **OnTaskRun** event is generated periodically during a file transfer while the background task is active. The arguments to this event are:

TaskId

An integer value that uniquely identifies the background task.

TimeElapsed

An integer value that specifies the amount of elapsed time in milliseconds.

Completed

An integer value that specifies an estimated percentage of completion.

The rate and number of times that this event will be generated depends on the task being performed. This event is generally analogous to the **OnProgress** event for file transfers that are performed in the current working thread, however the **OnTaskRun** event will occur for each individual background task that is active. The ***TimeElapsed*** parameter specifies the amount of time that the task has been active, and the ***Completed*** parameter specifies an estimated percentage of completion. This can be used to update the user interface if needed, however it is the application's responsibility to determine which UI component (such as a **ProgressBar** control) is associated with a particular task.

See Also

AsyncGetFile Method, AsyncPutFile Method, OnTaskBegin Event, OnTaskEnd Event

OnTimeout Event

The **OnTimeout** event is generated after an operation times out.

Syntax

Private Sub *object*_**OnTimeout**([*Index* **As Integer**])

Remarks

The **OnTimeout** event is generated after an operation times out. The amount of time that the component will wait for an operation to complete can be controlled by the **Timeout** property.

Example

```
Private Sub FileTransfer1_OnTimeout()
        lblStatus.Caption =  "Operation timed out"
End Sub
```

See Also

Timeout Property, OnCancel Event

File Transfer Protocol Control

Transfer files between a local and server and perform common file management functions on the server.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	FtpClientCtl.FtpClient
File Name	CSFTPX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.FtpClient.11
ClassID	42456D99-030C-4D03-A366-FA2975318C94
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 959, RFC 1579, RFC 2228

Overview

The File Transfer Protocol (FTP) control provides a comprehensive API which supports both high level operations, such as uploading or downloading files, as well as a collection of lower-level file I/O functions. In addition to file transfers, an application can create, rename and delete files and directories, search for files using wildcards and perform other common file management functions.

Files can be stored on the local file system or in memory, depending on the needs of your application and multiple file transfers be performed using a single function call. The control can also be used to manage files on the server and supports many of the common protocol extensions that can be used to access the remote file system. It understands a number of different directory listing formats, including those typically used with UNIX and Linux based systems, Windows server platforms, NetWare servers and VMS systems.

This control supports active and passive mode file transfers, firewall compatibility options, proxy servers and secure file transfers using the standard TLS and SFTP protocols. Secure file transfers support implicit and explicit TLS sessions, client certificates and up to 256-bit AES encryption.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop

and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

File Transfer Protocol Control Properties

Property	Description
Account	The account name for the current user
ActivePort	Gets and sets the range of ports used for active mode file transfers
AutoResolve	Determines if host names and IP addresses are automatically resolved
Blocking	Gets and sets the blocking state of the control
BufferSize	Gets and sets the size in bytes of an internal buffer that will be used during data transfers
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 client certificate
CertificatePassword	Gets and sets the password associated with the client certificate
CertificateStatus	Return the status of the server certificate
CertificateStore	Gets and sets the name of the client 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
ChannelMode	Gets and sets the security mode for the specified communications channel
CipherStrength	Return the length of the key used by the encryption algorithm
CodePage	Gets and sets the code page used for Unicode text conversion
DirectoryFormat	Gets and sets the current directory format type
Encoding	Gets and sets the character encoding that is used when a file name is sent to the server
Features	Gets and sets the features enabled for the current client session
FileMask	Gets and sets the current file mask
FileType	Gets and sets the current file transfer type
Fingerprint	Returns a string that uniquely identifies the server
HashStrength	Return the length of the message digest that was selected
HostAddress	Gets and sets the IP address of the server
HostName	Gets and sets the name of the server
IsBlocked	Return if the control is blocked performing an operation
IsConnected	Determine if the control is connected to a server
IsInitialized	Determine if the control has been initialized
IsReadable	Return if data can be read from the server without blocking
IsWritable	Return if data can be sent to the server without blocking
KeepAlive	Enable monitoring of the command channel to keep the client session active
LastError	Gets and sets the last error that occurred on the control

LastErrorString	Return a description of the last error to occur
Localize	Determines if remote file dates are localized to the current time zone
Options	Gets and sets the options that are used in establishing a connection
ParseList	Specify that file listings should be parsed by the control
Passive	Enable passive file transfers
Password	Gets and sets the password for the current user
Priority	Gets and sets the priority assigned to file transfers
PrivateKey	Gets and sets the private key file used for SSH authentication
ProxyHost	Gets and sets the host name of the proxy server
ProxyPassword	Gets and sets the proxy server password for the current user
ProxyPort	Gets and sets the port number for the proxy server
ProxyType	Gets and sets the current proxy server type
ProxyUser	Gets and sets the current proxy user name
RemoteFile	Gets and sets the file name specified in the current URL
RemotePath	Gets and sets the path specified in the current URL
RemotePort	Gets and sets the port number for a remote connection
ResultCode	Return the result code of the previous action
ResultString	Return a string describing the results of the previous action
Secure	Set or return if a connection to the server is secure
SecureCipher	Return the encryption algorithm used to establish the secure connection with the server
SecureHash	Return the message digest selected when establishing the secure connection with the server
SecureKeyExchange	Return the key exchange algorithm used to establish the secure connection with the server
SecureProtocol	Gets and sets the security protocol used to establish the secure connection with the server
System	Return information about the server
TaskCount	Return the number of active background file transfers
TaskId	Return the task ID for the last background file transfer
TaskList	Return the task ID for an active background file transfer
ThrowError	Enable or disable error handling by the container of the control
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
TransferBytes	Return the number of bytes transferred from the server
TransferBytesXL	Return the number of bytes transferred from the server
TransferRate	Return the current file transfer rate in bytes per second
TransferTime	Return the number of seconds elapsed during a data transfer

URL	Gets and sets the current URL used to access a file on the server
UserName	Gets and sets the current user name
Version	Return the current version of the object

Account Property

The account name for the current user.

Syntax

object.**Account** [= *account*]

Remarks

The **Account** property specifies the account name of the current user, if it is required by the server for authentication. Not all servers require an account name, in which case this property is ignored.

Data Type

String

See Also

Password Property, UserName Property, Connect Method, Login Method, Logout Method

ActivePort Property

Gets and sets the range of local port numbers used for active mode file transfers.

Syntax

object.**ActivePort**(*portrange*) [= *localport*]

Remarks

The **ActivePort** property property array is used to change the range of local port numbers used for active mode file transfers. The property array index specifies the port that should be changed, and may be one of the following values:

Value	Description
ftpActivePortLow	Change or return information for the low port number.
ftpActivePortHigh	Change or return information for the high port number.

The ***localport*** value specifies the new port number to be used. Valid port numbers are in the range of 1025 through 65535.

This property array is used to modify the range of local port numbers used for active mode file transfers. When using active mode, the client listens for an inbound connection from the server rather than establishing an outbound connection for the data transfer. In most cases, passive mode transfers are preferred because they mitigate potential compatibility issues with firewalls and NAT routers.

If active mode transfers are required, the default port range used when listening for the server connection is between 1024 and 5000. This is the standard range of ephemeral ports used by the Windows operating system. However, under some circumstances that range of ports may be too small, or a firewall may be configured to deny inbound connections on ephemeral ports. In that case, the **ActivePort** property can be used to specify a different range of port numbers.

While it is technically permissible to assign the low and high port numbers to the same value, effectively specifying a single active port number, this is not recommended as it can cause the transfer to fail unexpectedly if multiple file transfers are performed. A minimum range of at least 1000 ports is recommended. For example, if you specify a low port value of 40000 then it is recommended that the high port value be at least 41000. The maximum port value is 65535.

Data Type

Integer (Int32)

See Also

Features Property, Connect Method, Disconnect Method

AutoResolve Property

Determines if host names and IP 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 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 **AutoResolve** to False.

Data Type

Boolean

See Also

HostAddress Property, HostName Property

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 read data when none has been written, an error is generated. Events such as **OnConnect**, **OnDisconnect**, **OnRead** and **OnWrite** are only fired if the connection is non-blocking.

Data Type

Boolean

See Also

IsBlocked Property, IsReadable Property, IsWritable Property

BufferSize Property

Gets and sets the size in bytes of an internal buffer that will be used during data transfers.

Syntax

object.**BufferSize** [= *bytes*]

Remarks

Setting the **BufferSize** property specifies the size in bytes of an internal buffer that will be used during data transfers. Any set value greater than or equal to zero is acceptable. If the value is set to zero, then the default value of 4096 will be used. If the set value is between 1 and 255, inclusive, the buffer size will be set to 256. The maximum value is 1048576.

The speed of data transfers, particularly on uploads, may be sensitive to network type and configuration, and the size of the internal buffer used for data transfers. The default size of this buffer will result in good performance for a wide range of network characteristics. A larger buffer will not necessarily result in better performance. For example, a value of 1460, which is the typical Maximum Transmission Unit (MTU), may be optimal in many situations.

Data Type

Integer (Int32)

See Also

GetFile Method, PutFile Method

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 = FtpClient1.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 client certificate.

Syntax

object.**CertificateName** [= *name*]

Remarks

This property sets the common name or friendly name of the certificate that should be used to establish the connection with the server. 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. If a certificate name is specified, the certificate must have a private key associated with it, otherwise the connection attempt 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 client 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
stCertificateNone	No certificate information is available. A secure connection was not established with the server.
stCertificateValid	The certificate is valid.
stCertificateNoMatch	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.
stCertificateExpired	The certificate has expired and is no longer valid. The client can examine the **CertificateExpires** property to determine when the certificate expired.
stCertificateRevoked	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.
stCertificateUntrusted	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.
stCertificateInvalid	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:

```
FtpClient1.HostName = strHostName
FtpClient1.Secure = True
```

```
nError = FtpClient1.Connect()
If nError > 0 Then
     MsgBox "Unable to connect to server " & strHostName, vbExclamation
     Exit Sub
End If

If FtpClient1.CertificateStatus <> stCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          FtpClient1.Disconnect
          Exit Sub
     End If
End If

FtpClient1.Disconnect
```

See Also

CertificateExpires Property, CertificateIssued Property, CertificateIssuer Property, CertificateSubject Property, Secure Property

CertificateStore Property

Gets and sets the name of the client certificate store or file.

Syntax

object.**CertificateStore** [= *store*]

Remarks

This property sets the name of the certificate store that contains the client certificate that should be used when establishing a secure connection with 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 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

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 = FtpClient1.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 client certificate.

Syntax

object.**CertificateUser** [= *username*]

Remarks

This property sets the name of the user that owns the client certificate that will be used to establish a secure connection with the server. 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

ChannelMode Property

Gets and sets the security mode for the specified communications channel.

Syntax

object.**ChannelMode**(*channel*) [= *mode*]

Remarks

The **ChannelMode** property property array is used to change the security mode for either the command or data channel. The property array index specifies the channel that should be changed, and may be one of the following values:

Value	Description
ftpChannelCommand	Change or return information for the command channel. This is the communication channel used to send commands to the server and receive command result and status information from the server.
ftpChannelData	Change information for the data channel. This is the communication channel used to send or receive data during a file transfer.

The *mode* value specifies the new security mode for the specified channel. It may be one of the following values:

Value	Description
ftpChannelClear	Data sent and received on this channel should not be encrypted.
ftpChannelSecure	Data sent and received on this channel should be encrypted. Specifying this option requires that a secure connection has already been established with the server.

The **ChannelMode** property array is used to change the default mode for the specified channel, and is typically used to control whether or not data is encrypted during a file transfer. If a standard, non-secure connection has been established with the server, an error will be returned if you specify the **ftpChannelSecure** mode for either channel.

If you have established a secure connection and then specify the **ftpChannelClear** mode for the command channel, the client will send the CCC command to the server to indicate that commands should no longer be encrypted. If the server does not support this command, an error will be returned and the channel mode will remain unchanged. Once the command channel has been changed to clear mode, it cannot be changed back to secure mode. You must disconnect and re-connect to the server if you want to resume sending commands over an encrypted channel.

Changing the mode for the data channel requires that the server support the PROT command. If this command is not supported by the server, an exception will be thrown which must be handled by the application. You can only set a channel to secure mode if the **Secure** property is also set to True.

It is important to note that this property array should only be used after a connection has been established with the server. If you attempt to read the property or change a value prior to calling the **Connect** method, an exception will be thrown.

Data Type

Integer (Int32)

See Also

Features Property, Secure Property, Connect Method, Disconnect Method

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 converting text to and from Unicode.

Syntax

object.**CodePage** [= *value*]

Remarks

The **CodePage** property is an integer value which specifies how text is encoded. Any valid code page identifier may be specified. Some common values are:

Value	Description
Text sent and received using a string should be converted using the ANSI code page for the current locale.	
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.	
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.	
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.	
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.	
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 exchanged with an FTP server 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, with each character represented using 16 bits. When you send and receive data using the String data type, they will automatically be converted to a stream of bytes.

By default, strings are converted to an array of bytes using UTF-8 encoding, mapping the 16-bit

Unicode characters to 8-bit bytes. Similarly, when reading data 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 the text you receive appears to corrupted or characters are being replaced with question marks or other symbols, it is likely the file on the server is using a different character encoding. Most applications use UTF-8 encoding to represent non-ASCII characters; however, some text files may use a localized character set rather than using Unicode. Using the **GetText** and **PutText** methods in combination with this property will change how that text is converted to Unicode.

Strings are only guaranteed to be safe when sending and receiving text. Using a string data type is not recommended when uploading or downloading binary data. If possible, you should always use a byte array when using the **GetData** and **PutData** methods.

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.

Although strings in Visual Basic are internally managed as Unicode, the default common controls used in Visual Basic 6.0 do not support Unicode. Those controls, such as buttons, text boxes and labels, will automatically convert the Unicode text to ANSI using the current code page. This means that text in the end-user's native language (depending on system settings) may display correctly, although text in other languages using different character sets may not. Also note that the VB6 IDE is not Unicode aware and may display corrupted string values or invalid characters, such as with tooltip values when debugging.

For Unicode support in Visual Basic 6.0, it's recommended that you use third-party controls. An alternative that some developers have used is the Microsoft Forms 2.0 Object Library (FM20.DLL) that is part of Microsoft Office. It includes a collection of controls that support Unicode, however they are not redistributable and Microsoft has stated that their use with VB6 is unsupported.

Data Type

Integer (Int32)

See Also

FileType Property, GetData Method, GetText Method, PutData Method, PutText Method

DirectoryFormat Property

Gets and sets the current directory format type.

Syntax

object.**DirectoryFormat** [= *format*]

Remarks

The **DirectoryFormat** property specifies the format of a directory listing used by a FTP server. The directory format types supported are:

Value	Description
ftpDirectoryAuto	This value specifies that the control should automatically determine the format of the file lists returned by the server. It is recommended that most applications use this value and allow the control to automatically determine the appropriate file listing format used by the server.
ftpDirectoryUNIX	This value specifies that the server returns file lists in the format commonly used by UNIX servers. Note that many servers can be configured to return file listings in this format, even if they are not actually a UNIX based platform. Consult the technical reference documentation for your server for more information.
ftpDirectoryMSDOS	This value specifies that the server returns file lists in the format commonly used by MS-DOS based systems. This includes Windows IIS servers. Long file names will be returned if supported by the underlying filesystem, such as NTFS or FAT32.
ftpDirectoryVMS	This value specifies that the server returns file lists in the format commonly used by VMS servers. Note that VMS servers can be configured to return a standard UNIX style listing in additional to the default VMS format.
ftpDirectorySterling1	This value specifies that the server returns file listings in a proprietary format used by the Sterling server, which is used for EDI (Electronic Data Interchange) applications. This format uses a 13 byte status code.
ftpDirectorySterling2	This value specifies that the server returns file listings in a proprietary format used by the Sterling server, which is used for EDI (Electronic Data Interchange) applications. This format uses a 10 byte status code.
ftpDirectoryNetWare	This value specifies that the server returns file listings in a proprietary format used by NetWare servers. The format is similar to UNIX style listings except that file access and permissions are indicated by letter codes enclosed in brackets. This is the default format selected if the server identifies itself as a NetWare system.
ftpDirectoryMLSD	This value specifies that the server should return file listings in a machine-independent format as defined by RFC 3659. This format specifies file information as a sequence of name/value pairs, with the same format being used regardless of the operating system that the server is hosted on. Note that not all servers support this format, and some proxy servers may reject the command even if the remote server supports its use.

This property should only be set if the control cannot automatically determine the directory format

returned by the server. The default directory format is determined both by the server's operating system and by analyzing the format of the data returned by the server. If the control is unable to automatically determine the format, it will attempt to parse the list of files as though it is a UNIX style listing.

If this property is set to the default value **ftpDirectoryAuto** and the control can determine from the format of the file listing returned by the server, then the property will change value upon the first call to the **ReadDirectory** method or the first time the **FileList** event is generated.

Data Type

Integer (Int32)

See Also

System Property, ReadDirectory Method, FileList Event

Encoding Property

Gets and sets the character encoding that is used when a file name is sent to the server

Syntax

object.**Encoding** [= *flags*]

Remarks

The **Encoding** property returns one of the following integer values:

Value	Description
ftpEncodingANSI	File names are sent as 8-bit characters using the default character encoding for the current codepage. If the Unicode version of the functions are used, file names are converted from Unicode to ANSI using the current codepage before being sent to the server. This is the default encoding type.
ftpEncodingUTF8	File names that contain non-ASCII characters are sent using UTF-8 encoding. This encoding type is only available on servers that advertise support for UTF-8 encoding and permit that encoding type to be enabled by the client.

The **Encoding** property can be used to enable UTF-8 encoding of file names, which provides improved support for the use of international character sets. However, the server must provide support for UTF-8 encoding by advertising it in response to the FEAT command and it must support the OPTS command which is used to enable UTF-8 encoding. If the server does not advertise support for UTF-8, or the OPTS command fails with an error, then an exception will be thrown and the encoding type will not change.

Although it is possible to use the **Features** property to explicitly enable the **ftpFeatureUTF8** feature, this is not recommended. If the server has not advertised support for UTF-8 encoding in response to the FEAT command, that typically indicates that UTF-8 encoding is not supported. Attempting to force UTF-8 encoding can result in unpredictable behavior when file names contain non-ASCII characters.

It is important to note that not all FTP servers support UTF-8 encoding, and in some cases servers which advertise support for UTF-8 encoding do not implement the feature correctly. For example, a server may allow a client to enable UTF-8 encoding, but once enabled will not permit the client to disable it. Some servers may advertise support for UTF-8 encoding, however if the underlying file system does not support UTF-8 encoded file names, any attempt to upload or download a file may fail with an error indicating that the file cannot be found or created.

Data Type

Integer (Int32)

See Also

Features Property, Command Method

Features Property

Gets and sets the features enabled for the current client session.

Syntax

object.**Features** [= *flags*]

Remarks

The **Features** property returns a value which may be a combination of one or more of the following bit flags:

Value	Description	
&H00001	ftpFeatureSIZE	The server supports the SIZE command to determine the size of a file. If this feature is not enabled, the control will attempt to use the MLST or STAT command to determine the file size.
&H00002	ftpFeatureSTAT	The server supports using the STAT command to return information about a specific file. If this feature is not enabled, the client may not be able to obtain information about a specific file such as its size, permissions or modification time.
&H00004	ftpFeatureMDTM	The server supports the MDTM command to obtain information about the modification time for a specific file. This command may also be used to set the file time on the server.
&H00008	ftpFeatureREST	The server supports restarting file transfers using the REST command. If this feature is not enabled, the client will not be able to restart file transfers and must upload or download the complete file.
&H00010	ftpFeatureSITE	The server supports site specific commands using the SITE command. If this feature is not enabled, no site specific commands will be sent to the server.
&H00020	ftpFeatureIDLE	The server supports setting the idle timeout period using the SITE IDLE command to specify the number of seconds that the client may idle before the server terminates the connection.
&H00040	ftpFeatureCHMOD	The server supports modifying the permissions of a specific file using the SITE CHMOD command. If this feature is not enabled, the client will not be able to set the permissions for a file.
&H00080	ftpFeatureAUTH	The server supports explicit TLS sessions using the AUTH command. If this feature is not enabled, the client will only be able to connect to a secure server that uses implicit TLS connections. Changing this feature has no effect on standard, non-secure connections.
&H00100	ftpFeaturePBSZ	The server supports the PBSZ command which specifies

		the buffer size used with secure data connections. If this feature is disabled, it may prevent the client from changing the protection level on the data channel. Changing this feature has no effect on standard, non-secure connections.
&H00200	ftpFeaturePROT	The server supports the PROT command which specifies the protection level for the data channel. If this feature is disabled, the client will be unable to change the protection level on the data channel. Changing this feature has no effect on standard, non-secure connections.
&H00400	ftpFeatureCCC	The server supports the CCC command which returns the command channel to a non-secure mode. Changing this feature has no effect on standard, non-secure connections.
&H00800	ftpFeatureHOST	The server supports the HOST command which enables a client to specify the hostname after establishing a connection with a server that supports virtual hosting.
&H01000	ftpFeatureMLST	The server supports the MLST command which returns status information for files. If this feature is enabled, the MLST command will be used instead of the STAT command.
&H02000	ftpFeatureMFMT	The server supports the MFMT command which is used to change the last modification time for a file. If this command is supported, it is used instead of the MDTM command to change the modification time for a file.
&H04000	ftpFeatureXCRC	The server supports the XCRC command which returns the CRC-32 checksum for the contents of a specified file. This command is used for file verification.
&H08000	ftpFeatureMD5	The server supports the XMD5 command which returns an MD5 hash for the contents of a specified file. This command is used for file verification.
&H10000	ftpFeatureLANG	The server supports the LANG command which sets the language used for the current client session. Command responses and file naming conventions will use the specified language.
&H20000	ftpFeatureUTF8	The server supports the OPTS UTF-8 command which specifies UTF-8 encoding when specifying filenames. This feature is typically used in conjunction with setting the default language for the client session.
&H40000	ftpFeatureXQUOTA	The server supports the XQUOTA command which returns quota information for the current client session.
&H80000	ftpFeatureUTIME	The server supports the UTIME command which is used to change the last modification time for a specified file.

When a client connection is first established, the FEAT command is sent to the server to determine what features are available. However, as the client issues commands to the server, if the server reports that the command is unrecognized that feature will automatically be disabled in the client.

For example, the first time an application calls the **GetFileSize** method to determine the size of a file, the control will try to use the SIZE command. If the server reports that the SIZE command is not available, that feature will be disabled and the control will not use the command again during the session unless it is explicitly re-enabled. This is designed to prevent the control from repeatedly sending invalid commands to a server, which may result in the server aborting the connection.

Setting the **Features** property enables those features which have been specified. More than one feature may be enabled by combining the above constants using a bitwise Or operator. To test if a particular feature has been enabled, use the bitwise And operator. For example, in Visual Basic this can be done using the And and Or operators:

```
' If the SIZE command is enabled, disable it and make sure
' that the STAT command is enabled instead
If (FtpClient1.Features And ftpFeatureSIZE) <> 0 Then
    FtpClient1.Features = FtpClient1.Features And Not ftpFeatureSIZE
    FtpClient1.Features = FtpClient1.Features Or ftpFeatureSTAT
End If
```

Because features are specific to the current session, once you disconnect from the server they are reset. Even if you wish to reconnect to the same server, you must explicitly set the **Features** property again to those features which you wish to enable. Setting the **Features** property when the control is not connected to a server will cause the client session to only use those specified features for the next connection that is established. Setting the **Features** property during an active connection will change the features available for that session.

Data Type

Integer (Int32)

See Also

Encoding Property, Connect Method,

FileMask Property

Gets and sets the current file mask.

Syntax

object.**FileMask** [= *wildcard*]

Remarks

The **FileMask** property specifies the default wildcard mask to be used when uploading or downloading multiple files. The default value of an empty string indicates that all files in the specified directory should be uploaded or downloaded. Typically, this property is set to a wildcard mask that limits the files downloaded from the server to those which match a specific extension. For example, to download only those files that end in a ".dat" extension, the property could be set to the value "*.dat"

Note that the type of wildcards which may be used depend on the server and the type of file system that it is using. Take particular care when dealing with file systems that distinguish between upper- and lower-case letters in a filename.

Data Type

String

See Also

GetMultipleFiles Method, PutMultipleFiles Method

FileType Property

Gets and sets the current file transfer type.

Syntax

object.**FileType** [= *filetype*]

Remarks

The **FileType** property specifies the type of file transfer between the local and server. The file transfer types supported are:

Value	Description
ftpFileTypeAuto	The file type should be automatically determined based on the file name extension. If the file extension is unknown, the file type should be determined based on the contents of the file. The control has an internal list of common text file extensions, and additional file extensions can be registered using the **AddFileType** method.
ftpFileTypeASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings. The constant **ftpFileTypeText** is an alias for this value.
ftpFileTypeEBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.
ftpFileTypeImage	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executable programs. If the type of file cannot be automatically determined, it will always be considered a binary file. If this file type is specified when uploading or downloading text files, the native end-of-line character sequences will be preserved. The constant **ftpFileTypeBinary** is an alias for this value.
ftpFileTypeLocal	The file is a binary file that uses the local byte size for the server platform. On most servers, this file type is considered to be the same as **ftpFileTypeBinary**. Not all servers support this file type. It is recommended that you only specify this type if you know it is required by the server to transfer data correctly.

The file type should be set before a file is opened or created on the server. Once the file type is set, it is in effect for all files that are subsequently opened or created. Some methods, such as **OpenDirectory**, will temporarily change the default file type to **ftpFileTypeText** and then restore the current file type when they return.

Changing the value of this property has no practical effect when connected to an SFTP (SSH) server. They do not differentiate between text and binary files and the default file type will always be **ftpFileTypeBinary**. If your application is uploading or downloading a text file, this difference between FTP and SFTP is important because the operating system that hosts the server may have different end-of-line character conventions than the client system. For example, if you download a text file from a UNIX system using SFTP, the end-of-line is indicated by a single linefeed (LF) character However, on the Windows platform, the end-of-line is indicated by a carriage-return and linefeed sequence (CRLF).

If you are transferring binary data, you should always use **ftpFileTypeImage** and store the data in a **Byte** array using either the **GetData** or **PutData** methods. This will ensure that the data is sent or received exactly as-is without any character set or end-of-line conversion.

If you need to upload or download text stored in **String** variable, use the **GetText** and **PutText** methods. They will always set the file type to **ftpFileTypeText** and then restore the previous file type when the method returns. The value of the **CodePage** property will allow you to control how the text is converted to Unicode.

Data Type

Integer (Int32)

See Also

CodePage Property, AddFileType Method, GetData Method, GetFile Method, GetText Method, PutData Method, PutFile Method, PutText Method

Fingerprint Property

Returns a string that uniquely identifies the server.

Syntax

object.**Fingerprint**

Remarks

The **Fingerprint** property returns a string that consists of a series of hexadecimal values separated by colons. The value is unique to the server, and is an MD5 hash of the RSA host key. An application can use this value to determine if a connection has been established with the server previously by storing the server's host name, IP address and fingerprint in a file, registry key or a database.

Note that this property only returns a meaningful value after a secure connection has been established using the SSH protocol. For all other connections, it will return an empty string.

Data Type

String

See Also

Connect 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 server.

Syntax

object.**HostAddress** [= *ipaddress*]

Remarks

The **HostAddress** property can be used to set the IP address for a server 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, HostName Property

HostName Property

Gets and sets the name of the server.

Syntax

object.**HostName** [= *hostname*]

Remarks

The **HostName** property should be set to the name of the server 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

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.

Note that this property will return True if there is ***any*** blocking operation being performed by the application, regardless if the specified control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

Blocking Property, LastError Property

IsConnected Property

Determine if the control is connected to a server.

Syntax

object.**IsConnected**

Remarks

The **IsConnected** read-only property is set to a value of true if the control is connected with a server, otherwise the property has a value of false.

Data Type

Boolean

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

IsReadable Property

Return if data can be read from the server without blocking.

Syntax

object.**IsReadable**

Remarks

The **IsReadable** property returns True if data can be read from the server without blocking. For non-blocking connections, this property can be checked before the application attempts to read the data, preventing an error.

Data Type

Boolean

See Also

IsConnected Property, Read Method, OnRead Event

IsWritable Property

Return if data can be sent to the server without blocking.

Syntax

object.**IsWritable**

Remarks

The **IsWritable** property returns True if data can be written without blocking. For non-blocking connections, this property can be checked before the application attempts to send data to the server, 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 send data without an error. The next operation may result in an **stErrorOperationWouldBlock** or **stErrorOperationInProgress** error. The application must treat these errors as recoverable, and should be prepared to retry operations that result in them.

Data Type

Boolean

See Also

IsReadable Property, Write Method, OnWrite Event

KeepAlive Property

Enable monitoring of the command channel to keep the client session active.

Syntax

object.**KeepAlive** [= { True | False }]

Remarks

Setting the **KeepAlive** property to a value of true specifies that a background worker thread will be created to monitor the command channel and periodically send NOOP commands to the server if no commands have been sent recently. This can prevent the server from terminating the client connection during idle periods where no commands are being issued. However, it is important to keep in mind that many servers can be configured to also limit the total amount of time a client can be connected to the server, as well as the amount of time permitted between file transfers. If the server does not respond to the NOOP command, this option will be automatically disabled for the remainder of the client session.

It is recommended that you only enable this option if the connection to the server must be maintained for a relatively long period of time where there will be periods of inactivity. Never make the assumption that this option can prevent the server from terminating the connection. Most sites, particularly public FTP servers accessed over the Internet, have fairly restrictive policies designed to prevent clients from maintaining long-term connections. In most cases, if there are periods of time where your application will not be transferring files, it is more appropriate to disconnect from the server and then reconnect at a later point rather than attempting to hold the connection open.

The default value for this property is false.

Data Type

Boolean

See Also

Connect Method, GetFile Method, PutFile 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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

Localize Property

Determines if remote file dates are localized to the current timezone.

Syntax

object.**Localize** [= { True | False }]

Remarks

Setting the **Localize** property controls how remote file date and time values are localized when the **GetFileTime** method is called. If the property is set to True, then the file date and time will be adjusted to the current timezone. If the property is set to False, which is the default value, then the file date and time are returned as UTC (Coordinated Universal Time) values.

Data Type

Boolean

See Also

GetFileTime Method

Options Property

Gets and sets the options that are used in establishing a connection.

Syntax

object.**Options** [= *value*]

Remarks

The **Options** property is an integer value which specifies one or more options. The value specified for this property will be used as the default options when connecting to the server. The property value is created by using a bitwise operator with one or more of the following values:

Value	Description
ftpOptionPassive	This option specifies the client should attempt to establish a passive connection to the server. This means that instead of the client opening a port on the local system and waiting for the server to establish a connection back to the client, the client will establish a second data connection to the server. This mode is recommended for most systems that are behind a NAT router or firewall.
ftpOptionFirewall	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
ftpOptionNoAuth	This option specifies the server does not require authentication, or that it requires an alternate authentication method. When this option is used, the client

	connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.	
ftpOptionKeepAlive	This option specifies the client should attempt to keep the connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.	
&H10	ftpOptionNoAuthRSA	This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using TLS.
&H20	ftpOptionNoPwdNul	This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server. This option has no effect on standard or secure connections using TLS.
&H40	ftpOptionNoRekey	This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become

		non-responsive or abort the connection after being connected for an hour. This option has no effect on standard or secure connections using TLS.
&H80	ftpOptionCompatSID	This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using SSL.
&H100	ftpOptionCompatHMAC	This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions. This option has no effect on standard or secure connections using TLS.
&H200	ftpOptionVirtualHost	This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.
&H400	ftpOptionVerify	This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may

		occur when text files are uploaded or downloaded.
&H800	ftpOptionTrustedSite	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 the TLS protocol.
&1000	ftpOptionSecure	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
&H2000	ftpOptionSecureExplicit	This option specifies the client should use the AUTH TLS-P command to negotiate an explicit secure connection. Some servers may only require this when connecting to the server on ports other than 990.
&H4000	ftpOptionSecureShell	This option specifies the client should use the Secure Shell (SSH) protocol to establish the connection. This option will automatically be selected if the connection is established using port 22, the default port for SSH, and is only required if the server is configured to use a non-standard port number.
&H8000	ftpOptionSecureFallback	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.
&H10000	ftpOptionTunnel	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility

		features in the client.
&H20000	ftpOptionKeepAliveData	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the **ftpOptionKeepAlive** option and forces the client to continue to issue NOOP commands during the file transfer. This option only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
&40000	ftpOptionPreferIPv6	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.
&100000	ftpOptionHiResTimer	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
&200000	ftpOptionTLSReuse	This option specifies that TLS session reuse should be enabled for secure data connections. Some servers may require this option be enabled, although it should only used when required. This option is only valid for secure FTP (FTPS) connections and is not used with SFTP or secure HTTP connections. See the remarks below for more information.

Note that setting the **ftpOptionPassive** option is the same as setting the **Passive** property to True.

If the **ftpOptionSecureExplicit** option is specified, the client will first send an AUTH TLS command to the server. If the server does not accept this command, it will then send an AUTH SSL command. If both commands are rejected by the server, an explicit TLS session cannot be established. By default, both the command and data channels will be encrypted when a secure connection is established. To change this, set the **ChannelMode** property.

If the amount of time required to perform the transfer exceeds the server idle timeout period, the server may abort the connection. Using the **ftpOptionKeepAliveData** option tells the client to periodically send NOOP commands during the transfer to maintain the control connection. Some servers may not accept client commands while a transfer is in progress. You should only use this option if the server is timing out the control connection during large file transfers.

The **ftpOptionTLSReuse** option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms. This option is not compatible with servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627. To avoid potential problems with server compatibility, you should not specify this option for all FTP connections. It should only be used if specifically required by the server and your end-users should have the ability to selectively enable or disable this option.

Data Type

Integer (Int32)

Example

```
'
' The Ipswitch WS_FTP server accepts the AUTH command to establish
' an explicit TLS session on the default FTP port
'

FtpClient1.Options = ftpOptionSecureExplicit
nError = FtpClient1.Connect(strHostName)

'
' When the GlobalSCAPE Secure FTP server is configured in implicit
' authorization mode, it negotiates a secure session as soon as the
' connection is established and does not require a command
'

FtpClient1.Options = ftpOptionSecure
nError = FtpClient1.Connect(strHostName)
```

See Also

Priority Property, Secure Property, Connect Method

ParseList Property

Specify that file listings should be parsed by the control.

Syntax

object.**ParseList** [= { True | False }]

Remarks

The **ParseList** property is used to control how remote file lists are processed. If the property is set to False (the default value), then file lists are not processed by the control. The client application is responsible for reading through and parsing the list of files returned by the server. If the property is set to True, the control will parse the file list and generate a **FileList** event for each file in the list.

The control recognizes file listings in UNIX, MS-DOS and VMS formats, and will attempt to automatically determine the format that is being returned by the server. If the server does not return file lists in one of these formats, the **ParseList** property should be set to False, and the client application must parse the file listing itself.

Data Type

Boolean

See Also

FileList Method, GetFileStatus Method, FileList Event, LastFile Event

Passive Property

Enable passive file transfers.

Syntax

object.**Passive** [= {True | False}]

Remarks

The **Passive** property enables or disables passive file transfers between the local and server. In passive transfer mode, the client is responsible for establishing the data connection between the server and the local system. If the local system is behind a firewall or NAT router, it is recommended that you set this property to True.

Note that not all servers support passive file transfers. If the **Passive** property is set and the server does not support passive mode, an error will be returned the next time a file transfer is attempted.

Data Type

Boolean

See Also

Connect Method

Password Property

Gets and sets the password for the current user.

Syntax

object.**Password** [= *password*]

Remarks

The **Password** property specifies the password used to authenticate the user. The **UserName** and **Password** properties are used together to provide credentials to the server that will authenticate the client session. If these properties not set when the connection is established, the session will be anonymous. A server may allow a client to download files without authentication, but it is usually required when uploading files.

If you are connecting to a server using the SFTP (SSH) protocol and the server requires public/private key authentication, your application should set the **PrivateKey** property to the path of the user's private key. In this case, the **Password** property can be used to specify the password required to access the private key. If the private key was created without a password, this property should be set to an empty string.

Data Type

String

See Also

Account Property, PrivateKey Property, UserName Property, Connect Method, Login Method, Logout Method

Priority Property

Gets and sets a value which specifies the priority of file transfers.

Syntax

object.**Priority** [= *priority*]

Remarks

The **Priority** property can be used to control the processor usage, memory and network bandwidth allocated for file transfers. One of the following values may be specified:

Value	Description
ftpPriorityBackground	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
ftpPriorityLow	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
ftpPriorityNormal	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
ftpPriorityHigh	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
ftpPriorityCritical	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

The **ftpPriorityNormal** priority balances resource utilization and transfer speed while ensuring that a single-threaded application remains responsive to the user. Lower priorities reduce the overall resource utilization at the expense of transfer speed. For example, if you create a worker thread to download a file in the background and want to ensure that it has a minimal impact on the process, the **ftpPriorityBackground** value can be used.

Higher priority values increase the memory allocated for the transfers and increases processor utilization for the transfer. The **ftpPriorityCritical** priority maximizes transfer speed at the expense of system resources. It is not recommended that you increase the file transfer priority unless you understand the implications of doing so and have thoroughly tested your application. If the file transfer is being performed in the main UI thread, increasing the priority may interfere with the normal processing of Windows messages and cause the application to appear to become non-responsive. It is also important to note that when the priority is set to **ftpPriorityCritical**, normal progress events will not be generated during the transfer.

Data Type

Integer (Int32)

See Also

GetData Method GetFile Method PutData Method PutFile Method

PrivateKey Property

Gets and sets the private key file used for SSH authentication.

Syntax

object.**PrivateKey** [= *filename*]

Remarks

The **PrivateKey** property specifies the path to the private key file used to authenticate the user. The private key is used in combination with the value of the **UserName** property to provide credentials to an SSH server. If public/private key authentication is not required by the server, this property is should be set to an empty string. This property is only used with SFTP connections and is ignored for standard FTP connections or secure connections using FTPS (FTP+TLS).

The **PrivateKey** property value can use environment variables enclosed in percent symbols, and the path to the private key file will be normalized. It is recommended you always use an absolute path to the private key file. If you do not include a path, it will use the current working directory for the process. This can produce inconsistent results because the current working directory for a process is a global value and it can be changed at any time.

The private key file name must resolve to a text file which can be read by the current process. If the file does not exist, or it does not specify a PEM formatted file which contains an RSA or OpenSSH private key, the connection will fail and the last error code will be set to `stErrorInvalidPrivateKey`.

Data Type

String

Example

```
' Connect to the server using a private key for authentication
FtpClient1.HostName = "file.server.tld"
FtpClienn1.RemotePort = 22
FtpClient1.UserName = "username"
FtpClient1.PrivateKey = "%USERPROFILE%\\.ssh\\id_rsa.pem"
FtpClient1.Options = ftpOptionSecureShell Or ftpOptionKeepAlive

nError = FtpClient1.Connect()
If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

In this example, the path to the private key file will be expanded to an absolute path because the USERPROFILE environment variable defines the home directory for the current user.

See Also

Account Property, Password Property, UserName Property, Connect Method

ProxyHost Property

Gets and sets the host name of the proxy server.

Syntax

object.**ProxyHost** [= *hostname*]

Remarks

The **ProxyHost** property should be set to the name of the proxy server that you want to connect to. This property may be set to either a fully qualified domain name, or an IP address. This property is only used if the **ProxyType** property is set to a non-zero value.

Data Type

String

See Also

HostAddress Property, Password Property, ProxyPassword Property, ProxyPort Property, ProxyType Property, ProxyUser Property, UserName Property, Connect Method

ProxyPassword Property

Gets and sets the proxy server password for the current user.

Syntax

object.**ProxyPassword** [= *password*]

Remarks

The **ProxyPassword** property specifies the password used to authenticate the user to the proxy server. If a password is not required by the server, this property is ignored.

Data Type

String

See Also

HostAddress Property, Password Property, ProxyHost Property, ProxyPort Property, ProxyType Property, ProxyUser Property, UserName Property, Connect Method

ProxyPort Property

Gets and sets the port number for the proxy server.

Syntax

object.**ProxyPort** [= *portno*%]

Remarks

The **ProxyPort** property is used to set the port number that the control will use to establish a connection with the proxy server. A value of zero specifies that the client will connect to the proxy server using the standard FTP service port.

Data Type

Integer (Int32)

See Also

HostAddress Property, Password Property, ProxyHost Property, ProxyPassword Property, ProxyType Property, ProxyUser Property, UserName Property, Connect Method

ProxyType Property

Gets and sets the current proxy server type.

Syntax

object.**ProxyType** [= *proxytype*]

Remarks

The **ProxyType** property specifies the type of proxy server that the client is connecting to. The supported proxy server types are as follows:

Value	Description
ftpProxyTypeNone	No proxy server is being used. This is the default value.
ftpProxyTypeUser	The client is not logged into the proxy server. The USER command is sent in the format username@ftpsite followed by the password. This is the format used with the Gauntlet proxy server.
ftpProxyTypeLogin	The client is logged into the proxy server. The USER command is then sent in the format username@ftpsite followed by the password. This is the format used by the InterLock proxy server.
ftpProxyTypeOpen	The client is not logged into the proxy server. The OPEN command is sent specifying the host name, followed by the username and password.
ftpProxyTypeSite	The client is logged into the server. The SITE command is sent, specifying the host name, followed by the username and the password.
ftpProxyTypeOther	This special proxy type specifies that another, undefined proxy server is being used. The client connects to the proxy host, but does not attempt to authenticate the client. The application is responsible for negotiating with the proxy server, typically using the Command property to send specific command sequences.

Data Type

Integer (Int32)

See Also

HostAddress Property, Password Property, ProxyHost Property, ProxyPassword Property, ProxyPort Property, ProxyUser Property, UserName Property, Connect Method

ProxyUser Property

Gets and sets the current proxy user name.

Syntax

object.**ProxyUser** [= *username*]

Remarks

The **ProxyUser** property specifies the user that is logging in to the proxy server. If the proxy server does not require the user to login, then this property is ignored.

Data Type

String

See Also

HostAddress Property, Password Property, ProxyHost Property, ProxyPassword Property, ProxyPort Property, ProxyType Property, UserName Property, Connect Method

RemoteFile Property

Sets or returns the file name specified in the current URL.

Syntax

object.**RemoteFile** [= *value*]

Remarks

The **RemoteFile** property returns the name of the file that was specified when the **URL** property was set. Changing the value of this property causes the URL to change. Note that the control does not check to make sure that the remote file name is valid or that it exists on the server, it is simply a string that is part of the FTP URL.

Data Type

String

See Also

RemotePath Property, URL Property

RemotePath Property

Sets or returns the path specified in the current URL.

Syntax

object.**RemotePath** [= *value*]

Remarks

The **RemotePath** property returns the path that was specified when the **URL** property was set. Changing the value of this property causes the URL to change. Note that the control does not check to make sure that the remote path is valid or that it exists on the server, it is simply a string that is part of the FTP URL.

Note that the path is relative to the user's home directory and should not be considered an absolute path from the root directory on the server. If no username and password is provided, then an anonymous session is used and the path is relative to the public directory used by the FTP server.

Data Type

String

See Also

RemoteFile Property, URL Property

RemotePort Property

Gets and sets the port number for a remote connection.

Syntax

object.**RemotePort** [= *portnumber*]

Remarks

The **RemotePort** property is used to set the port number that the control will use to establish a connection with the server.

Data Type

Integer (Int32)

See Also

HostAddress Property, HostName Property

ResultCode Property

Return the result code of the previous action.

Syntax

object.**ResultCode**

Remarks

The **ResultCode** read-only property returns the result code of the last action performed by the client. This property should be checked after the **Command** method is used to execute a command on the server to determine if the operation was successful. Result codes are three-digit numeric values returned by the server and may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

It is important to note that while some result codes have become standardized, not all servers respond to commands using the same result codes. For example, one server may respond with a result code of 221 to indicate success, while another may respond with a value of 235. It is recommended that applications check for ranges of values to determine if a command was successful, not a specific value.

Data Type

Integer (Int32)

See Also

ResultString Property, Command Method, OnCommand Event

ResultString Property

Return a string describing the results of the previous action.

Syntax

object.**ResultString**

Remarks

The **ResultString** read-only property returns the result string from the last action taken by the client. This string is generated by the server, and typically is used to describe the result code. For example, if an error is indicated by the result code, the result string may describe the condition that caused the error.

Data Type

String

See Also

ResultCode Property, Command Method, OnCommand Event

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 must 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 strongly recommended that any application that sets this property True use error handling to trap an errors that may occur. If the control 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.

Data Type

Boolean

Example

The following example establishes a secure connection to a server and retrieves a file:

```
FtpClient1.HostName = strHostName
FtpClient1.RemotePort = 21
FtpClient1.UserName = strUserName
FtpClient1.Password = strPassword
FtpClient1.Secure = True

nError = FtpClient1.Connect()
If nError > 0 Then
    MsgBox "Unable to connect to server " & strHostName, vbExclamation
    Exit Sub
End If

If FtpClient1.CertificateStatus <> stCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          FtpClient1.Disconnect
          Exit Sub
     End If
End If

nError = FtpClient1.GetFile(strLocalFile, strRemoteFile)
FtpClient1.Disconnect

If nError > 0 Then
    MsgBox "Unable to download " & strRemoteFile, vbExclamation
    Exit Sub
End If
```

See Also

CertificateStatus 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
stCipherNone	No cipher has been selected. This is not a secure connection with the server.
stCipherRC2	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.
stCipherRC4	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.
stCipherRC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
stCipherDES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher using 56-bit keys.
stCipherDES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively using a 168-bit key length.
stCipherDESX	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.
stCipherAES	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.
stCipherSkipjack	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
stCipherBlowfish	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
stHashMD5	The MD5 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA1	The SHA-1 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA256	The SHA-256 algorithm has been selected.
stHashSHA384	The SHA-384 algorithm has been selected.
stHashSHA512	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
stKeyExchangeNone	No key exchange algorithm has been selected. This is not a secure connection with the server.
stKeyExchangeRSA	The RSA public key exchange algorithm has been selected.
stKeyExchangeKEA	The KEA public key exchange algorithm has been selected. This is an improved version of the Diffie-Hellman public key algorithm.
stKeyExchangeDH	The Diffie-Hellman public key exchange algorithm has been selected.
stKeyExchangeECDH	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

System Property

Return information about the server.

Syntax

object.**System**

Remarks

The **System** property returns information about the server operating system. This is a read-only property that can be used by the application to identify the type of server that the client has connected to. Reading this property will cause the SYST command to be sent to the server and will only return a useful value after a connection has been established with the server.

By convention, the first whitespace separated token in the string identifies the general operating system platform. For example, here are some strings commonly returned by various FTP servers:

Example	Description
UNIX Type: L8	A standard UNIX based server. This is the most common value returned by servers, and this indicates that the server supports UNIX file naming and directory listing conventions. This string may also include additional information such as the specific variant of UNIX and its version. The L8 portion of the string is a convention that lets the client know that a byte consists of 8 bits.
Windows_NT Version 5.0	A standard Windows based server, typically part of Internet Information Services (ISS). The server will use Windows file naming and directory listing conventions. The version identifies the specific release of Windows. For example, version 4.0 specifies Windows NT 4.0 and 5.0 specifies Windows 2000.
VMS V7.1 AlphaServer	A server running the VMS operating system. The server will use the standard file naming and directory listing conventions for that platform. Note that it is possible that a VMS system may also be configured to operate in a UNIX emulation mode, in which case it will return UNIX instead of VMS.
NetWare system type	A server running the NetWare operating system. The server will use the standard file naming and directory listing conventions for that platform. Note that it is possible that a NetWare system may be configured to operate in a UNIX emulation mode, in which case it return UNIX instead of NetWare.
WORLDGROUP Type: L8	A server running the WorldGroup software on the Windows platform. This server supports UNIX file naming and directory listing conventions. WorldGroup is a collaborative workgroup, email and file sharing service which includes an FTP server.

Data Type

String

See Also

DirectoryFormat Property, IsConnected Property, Connect Method, Disconnect Method

TaskCount Property

Return the number of active background file transfers.

Syntax

object.**TaskCount**

Remarks

The **TaskCount** property returns the number of background file transfers that are currently in progress. One common use for this property is to create a timer that periodically checks this value when a series of background transfers are started. When the property returns a value of zero, that indicates all of the background transfers have completed. This property can also be used to enumerate the active background tasks in conjunction with the **TaskList** property.

Data Type

Integer (Int32)

See Also

TaskList Property, AsyncGetFile Method, AsyncPutFile Method, TaskAbort Method, TaskWait Method

TaskId Property

Return the task ID for the last background file transfer.

Syntax

object.**TaskId**

Remarks

The **TaskId** property returns the task ID associated with the last background task that started. The value of this property is only meaningful after the **AsyncGetFile** or **AsyncPutFile** method is called to initiate a background file transfer, and the value will change with each subsequent background transfer that is performed. If this property returns a value of zero, that indicates that no background tasks have been started for this instance of the control.

To enumerate the active background tasks, use the **TaskCount** property and the **TaskList** property array.

Data Type

Integer (Int32)

See Also

TaskCount Property, TaskList Property, AsyncGetFile Method, AsyncPutFile Method, TaskAbort Method, TaskWait Method

TaskList Property

Return the task ID for an active background file transfer.

Syntax

object.**TaskList**(*Index*)

Remarks

The **TaskList** property is a zero-based array that returns an ID associated with an active background task. The current number of active tasks can be determined using the **TaskCount** property. If the index value specified for this property array exceeds the number of active tasks, an exception will be thrown.

As background tasks complete and additional tasks are started, the values returned by this property array will change. The application should never make an assumption about the actual task ID values returned or the order they are returned. While task IDs are assigned sequentially, they should be considered opaque values that are unique to the process. When a background task completes, its corresponding task ID is removed from the list of active tasks and this can potentially change the task ID values associated with each index into the property array.

Data Type

Integer (Int32)

See Also

TaskCount Property, TaskId Property, AsyncGetFile Method, AsyncPutFile Method, TaskAbort Method, TaskWait Method

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
FtpClient1.ThrowError = False
nError = FtpClient1.Connect(strHostName)

If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

FtpClient1.ThrowError = True
FtpClient1.Connect strHostName

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error.

Note that the **Timeout** property also determines the amount of time the control will spend attempting to connect to a server. If a connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
ftpTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
ftpTraceError	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.
ftpTraceWarning	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.
ftpTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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

TransferBytes Property

Return the number of bytes transferred from the server.

Syntax

object.**TransferBytes**

Remarks

The **TransferBytes** property returns the number of bytes that have been copied to or from the FTP server. If this property is read while a transfer is ongoing, the property returns the number of bytes that have been copied up to that point. If read after a transfer has completed, the total number of bytes copied is returned.

If the value would exceed 2,147,483,647 bytes (the maximum value for a 32-bit integer) this property will return -1 to indicate an overflow condition. If you are potentially transferring files larger than 2 GiB in size, you should use the **TransferBytesXL** property instead, which returns the number of bytes as a **Double** floating-point value.

This property value is reset with every data transfer.

Data Type

Integer (Int32)

See Also

TransferBytesXL Property, TransferRate Property, TransferTime Property, OnProgress Event

TransferBytesXL Property

Return the number of bytes transferred from the server.

Syntax

object.**TransferBytesXL**

Remarks

The **TransferBytesXL** property returns the number of bytes that have been copied to or from the FTP server. This property returns the number of bytes as a **Double** floating-point value instead of a **Long** integer, making it suitable for very large files that exceed 2 GiB in size.

If this property is read while a transfer is ongoing, the property returns the number of bytes that have been copied up to that point. If read after a transfer has completed, the total number of bytes copied is returned.

This property value is reset with every data transfer.

Data Type

Double

See Also

TransferRate Property, TransferTime Property, OnProgress Event

TransferRate Property

Return the current file transfer rate in bytes per second.

Syntax

object.**TransferRate**

Remarks

The **TransferRate** property returns the rate at which the file data is being transferred, expressed in bytes per second. If this property is read while a transfer is ongoing, it returns the current average transfer rate.

If this property is read after the transfer has completed, it returns the final transfer rate which is calculated as the total number of bytes transferred divided by the number of seconds to complete the transfer. This property value is reset with every data transfer.

Data Type

Integer (Int32)

See Also

TransferBytes Property, TransferTime Property, OnProgress Event

TransferTime Property

Return the number of seconds elapsed during a data transfer.

Syntax

object.**TransferTime**

Remarks

The **TransferTime** property returns the number of seconds that have elapsed since the last data connection was opened on the server. If the property is read while a transfer is ongoing, it returns the current elapsed time since the file transfer started.

If the property is read after the transfer has completed, it returns the total number of seconds it took to transfer the file. This property value is reset with every data transfer.

Data Type

Integer (Int32)

See Also

TransferBytes Property, TransferRate Property, OnProgress Event

URL Property

Gets and sets the current URL used to access a file on the server.

Syntax

object.**URL** [= *url*]

Remarks

The **URL** property returns the current Uniform Resource Locator string which is used by the control to access a file on the server. URLs have a specific format which provides information about the server, port, path and file name, as well as optional information such as a username and password for authentication:

```
[ftp|ftps|sftp]://[username:
[password]@]hostname[:port]/[path/...]filename[;type=id]
```

The first part of the URL is the scheme, and in this case will always be "ftp", "ftps" or "sftp" depending on which protocol is required. If a username and password is required for authentication, then this will be included in the URL before the name of the server; otherwise an anonymous FTP session is assumed. Next, there is the name of the server to connect to, optionally followed by a port number. If no port number is given, then the default port for the protocol will be used. This is followed by the path, and then the name of the file on the server. An optional file type may be specified as well, with the type identifier being either "a" for text files or "i" for binary files.

One important consideration when using FTP URLs is that the path is relative to the user's home directory and should not be considered an absolute path from the root directory on the server. If no username and password is provided, then an anonymous session is used and the path is relative to the public directory used by the FTP server.

Here are some common examples of URLs used to access files on an FTP server:

```
ftp://www.example.com/pub/financial/jan2023.xlsx
```

In this example, the server is www.example.com, the path is "pub/financial" and the file name is "jan2023.xlsx". The default port will be used to access the file, and no username and password is provided for authentication so this file must be publicly available to anonymous users.

```
ftp://www.example.com:2121/employees/picnic.docx
```

In this example, the server is www.example.com, the path is "employees" and the file name is "picnic.docx". However, the client should connect to an alternative port number, in this case 2121. This file must also be available to anonymous users because no username or password has been specified.

```
ftps://executive:secret@www.example.com/corporate/projections/sales2024.xlsx
```

In this example, the server is www.example.com and, the path is "corporate/projections" and the file name is "sales2024.xlsx". Because the protocol is ftps, a secure connection on port 990 will be established. The user name "executive" and password "secret" will be used to authenticate the session.

When setting the **URL** property, the control will parse the string and automatically update the **HostName**, **RemotePort**, **UserName**, **Password**, **RemotePath** and **RemoteFile** properties according to the values specified in the URL. This enables an application to simply provide the URL and then call the **Connect** method to establish the connection.

Note that if this property is assigned a value which cannot be parsed, the control will throw an error that indicates that the property value is invalid. In a language like Visual Basic it is important that you implement an error handler, particularly if you are assigning a value to the property based on user input. If the user enters an invalid URL and there is no error handler, it could result in an exception which terminates the application.

Data Type

String

Example

```
' Setup error handling since the control will throw an error
' if an invalid URL is specified

On Error Resume Next: Err.Clear
FtpClient1.URL = Text1.Text

' Check the Err object to see if an error has occurred, and
' if so, let the user know that the URL is invalid

If Err.Number <> 0 Then
    MsgBox "The specified URL is invalid", vbExclamation
    Text1.SetFocus
    Exit Sub
End If

' Reset error handling and connect to the server using the
' default property values that were updated when the URL
' property was set (ie: HostName, RemotePort, UserName, etc.)

On Error GoTo 0
nError = FtpClient1.Connect()

If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' Change to the directory specified by the RemotePath property

nError = FtpClient1.ChangeDirectory(FtpClient1.RemotePath)
If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' Download the file to the local system
strLocalFile = strLocalPath & "\" & FtpClient1.RemoteFile
nError = FtpClient1.GetFile(FtpClient1.RemoteFile, strLocalFile)
```

See Also

HostAddress Property, HostName Property, Password Property, RemoteFile Property, RemotePath Property, RemotePort Property, UserName Property, Connect Method

UserName Property

Gets and sets the current user name.

Syntax

object.**UserName** [= *username*]

Remarks

The **UserName** property specifies the user that is logging in to the server, and is required for authentication purposes.

Data Type

String

See Also

Account Property, Password Property, PrivateKey Property, Connect Method, Login Method, Logout Method

Version Property

Return the current version of the object.

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

File Transfer Protocol Control Methods

Method	Description
AddFileType	Associate a file name extension with a specific file type
AsyncGetFile	Download a file from the server to the local system in the background
AsyncPutFile	Upload a file from the local system to the server in the background
Cancel	Cancels the current blocking network operation
ChangeDirectory	Return data read from the server
CloseDirectory	Close the directory that was opened for reading on the server
CloseFile	Close the remote file that was opened for reading or writing
Command	Send a custom command to the server
Connect	Establish a connection with a server
CreateFile	Create a new file or overwrite an existing file
DeleteFile	Delete a file on the server
Disconnect	Terminate the connection with a server
GetData	Download the contents of a file and return it in the specified buffer
GetDirectory	Return the current working directory on the server
GetFile	Download a file from the server to the local system
GetFileList	Return an unparsed list of files in the specified directory
GetFilePermissions	Return the access permissions for a file on the remote system
GetFileSize	Returns the size of the specified file on the server
GetFileStatus	Return status information about a specific file
GetFileTime	Returns the modification date and time for specified file on the server
GetFileType	Returns the file type for a file on the local system
GetMultipleFiles	Download multiple files from the server to the local system
GetText	Download the contents of a text file and return it in a string buffer
Initialize	Initialize the control and validate the runtime license key
Login	Login to the server
Logout	Log the current user off the server
MakeDirectory	Create a new directory on the server
OpenDirectory	Open the specified directory on the server for reading
OpenFile	Open an existing file or creates a new file on the server
PutData	Upload the contents of buffer and store it in a file on the server
PutFile	Upload a file from the local system to the server

PutMultipleFiles	Upload multiple files from the local system to the server
PutText	Upload the contents of a string buffer and store it in a text file on the server
Read	Return data read from the server
ReadDirectory	Read a directory entry from the server
RemoveDirectory	Remove a directory on the server
RenameFile	Change the name of a file on the server
Reset	Reset the internal state of the control
SetFilePermissions	Change the access permissions for a file on the server
SetFileTime	Changes the modification date and time for a file on the server
TaskAbort	Abort the specified asynchronous task
TaskDone	Determine if an asynchronous task has completed
TaskResume	Resume execution of an asynchronous task
TaskSuspend	Suspend execution of an asynchronous task
TaskWait	Wait for an asynchronous task to complete
Uninitialize	Uninitialize the control and release any system resources that were allocated
VerifyFile	Compare the contents of a local file against a file stored on the server
Write	Write data to the server

AddFileType Method

Associate a file name extension with a specific file type.

Syntax

object.**AddFileType**(*FileExtension*, *FileType*)

Parameters

FileExtension

A string that specifies the file name extension.

FileType

Specifies the type of file associated with the file extension. This parameter can be one of the following values:

Value	Description
ftpFileTypeASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings. The constant **ftpFileTypeText** is an alias for this value.
ftpFileTypeEBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.
ftpFileTypeImage	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executable programs. If the type of file cannot be automatically determined, it will always be considered a binary file. If this file type is specified when uploading or downloading text files, the native end-of-line character sequences will be preserved. The constant **ftpFileTypeBinary** is an alias for this 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.

Remarks

The **AddFileType** method is used to associate specific file types with file name extensions. The control has an internal list of standard text file extensions which it automatically recognizes. This method can be used to extend or modify that list for the client session.

See Also

FileType Property, GetFile Method, GetFileType Method, PutFile Method

AsyncGetFile Method

Download a file from the server to the local system in the background.

Syntax

object.**AsyncGetFile**(*LocalFile*, *RemoteFile*, [*Options*], [*Offset*])

Parameters

LocalFile

A string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

RemoteFile

A string that specifies the file on the server that will be transferred to the local system. The file pathing and name conventions must be that of the server.

Options

A numeric bitmask which specifies one or more options. This argument may be any one of the following values:

Value	Description
ftpTransferDefault	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the remote file. If the Options argument is omitted, this is the transfer mode which will be used.
ftpTransferAppend	This option specifies that if the local file exists, the contents of file on the server is appended to the local file. If the local file does not exist, it is created.

Offset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

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 **AsyncGetFile** method will download the contents of a remote file to a file on the local system. It is similar to the **GetFile** method, however it retrieves the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being downloaded from the server. This method requires that you explicitly establish a connection using the **Connect** method. All background tasks will duplicate the active connection and use it establish a secondary connection with the server to perform the file transfer. If you wish to perform multiple asynchronous file transfers from different servers, you must create an instance of the control for each server.

After this method is called, the **OnTaskBegin** event will be fired, indicating that the background task has begun the process of connecting to the server and performing the file transfer. As the file is

downloaded, the control will periodically invoke the **OnTaskRun** event handler. When the transfer has completed, the **OnTaskEnd** event will be fired. It is not required that you implement handlers for these events.

To determine when a transfer has completed without implementing any event handlers, periodically call the **TaskDone** method. If you wish to block the current thread and wait for the transfer to complete, call the **TaskWait** method. To stop a background file transfer that is in progress, call the **TaskAbort** method. This will signal the background worker thread to cancel the transfer and terminate the session.

This method can be called multiple times to download more than one file in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. The application should not make any assumptions about the sequence in which background transfers are performed or the order in which they may complete.

Example

```
' Establish a connection to the server
nError = FtpClient1.Connect(strHostName, 21, strUserName, strPassword)

If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' Download a file in the background
nError = FtpClient1.AsyncGetFile(strLocalFile, strRemoteFile)

If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

TaskId Property, AsyncPutFile Method, TaskAbort Method, TaskDone Method, TaskWait Method, OnTaskBegin Event, OnTaskEnd Event, OnTaskRun Event

AsyncPutFile Method

Upload a file from the local system to the server in the background.

Syntax

object.**AsyncPutFile**(*LocalFile*, *RemoteFile*, [*Options*], [*Offset*])

Parameters

LocalFile

A string that specifies the file on the local system that will be transferred to the server. The file pathing and name conventions must be that of the local host.

RemoteFile

A string that specifies the file on the server that will be created, overwritten or appended to. The file pathing and name conventions must be that of the server.

Options

A numeric bitmask which specifies one or more options. This argument may be any one of the following values:

Value	Description
ftpTransferDefault	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the local file. If the Options argument is omitted, this is the transfer mode which will be used.
ftpTransferAppend	This option specifies that if the remote file exists, the contents of file on the local system is appended to the remote file. If the remote file does not exist, it is created.

Offset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

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 **AsyncPutFile** method will upload the contents of a file on the local system to the server. It is similar to the **PutFile** method, however it retrieves the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being uploaded to the server. This method requires that you explicitly establish a connection using the **Connect** method. All background tasks will duplicate the active connection and use it establish a secondary connection with the server to perform the file transfer. If you wish to perform multiple asynchronous file transfers from different servers, you must create an instance of the control for each server.

After this method is called, the **OnTaskBegin** event will be fired, indicating that the background task has begun the process of connecting to the server and performing the file transfer. As the file is

uploaded, the control will periodically invoke the **OnTaskRun** event handler. When the transfer has completed, the **OnTaskEnd** event will be fired. It is not required that you implement handlers for these events.

To determine when a transfer has completed without implementing any event handlers, periodically call the **TaskDone** method. If you wish to block the current thread and wait for the transfer to complete, call the **TaskWait** method. To stop a background file transfer that is in progress, call the **TaskAbort** method. This will signal the background worker thread to cancel the transfer and terminate the session.

This method can be called multiple times to upload more than one file in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. The application should not make any assumptions about the sequence in which background transfers are performed or the order in which they may complete.

Example

```
' Establish a connection to the server
nError = FtpClient1.Connect(strHostName, 21, strUserName, strPassword)

If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' Upload a file in the background
nError = FtpClient1.AsyncPutFile(strLocalFile, strRemoteFile)

If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

TaskId Property, AsyncGetFile Method, TaskAbort Method, TaskDone Method, TaskWait Method, OnTaskBegin Event, OnTaskEnd Event, OnTaskRun Event

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

Disconnect Method, Reset Method, OnCancel Event

ChangeDirectory Method

Change the current working directory on the server.

Syntax

object.**ChangeDirectory**(*RemotePath*)

Parameters

RemotePath

A string which specifies the directory on the server. The file naming conventions must be that of the host operating system.

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 **ChangeDirectory** method changes the current working directory on the server. This function uses the CWD command to change the current working directory. The user must have the appropriate permission to access the specified directory.

See Also

CloseDirectory Method, GetDirectory Method, OpenDirectory Method, ReadDirectory Method

CloseDirectory Method

Close the directory that was opened for reading on the server.

Syntax

object.**CloseDirectory**

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 **CloseDirectory** method closes the directory that was opened on the server using the **OpenDirectory** method. This method must be called once all of the files have been read from the server, otherwise an error will be returned on all subsequent attempts to transfer files or read other directories.

See Also

OpenDirectory Method, ReadDirectory Method

CloseFile Method

Close the remote file that was opened for reading or writing.

Syntax

object.**CloseFile**

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 **CloseFile** method closes the file that was created using the **CreateFile** method, or opened for reading using the **OpenFile** method.

See Also

CreateFile Method, OpenFile Method

Command Method

Send a custom command to the server.

Syntax

object.**Command**(*Command*, [*Parameters*], [*Options*])

Parameters

Command

A string which specifies the command to send. Valid commands vary based on the Internet protocol and the type of server that the client is connected to. Consult the protocol standard and/or the technical reference documentation for the server to determine what commands may be issued by a client application.

Parameters

An optional string which specifies one or more parameters to be sent along with the command. If more than one parameter is required, most Internet protocols require that they be separated by a single space character. Consult the protocol standard and/or technical reference documentation for the server to determine what parameters should be provided when issuing a specific command. If no parameters are required for the command, this argument may be omitted.

Options

A numeric value which specifies one or more options. Currently this argument is reserved and should either be omitted, or a value of zero should always be used.

Return Value

A value of zero is returned if the command was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure. To determine the result code returned by the server in response to the command, read the value of the **ResultCode** property.

Remarks

The **Command** method sends a command to the server and processes the result code sent back in response to that command. This method can be used to send custom commands to a server to take advantage of features or capabilities that may not be supported internally by the control.

This method should only be used when the application needs to send a custom, site-specific command or send a command that is not directly supported by the control. This method should never be used to issue a command that opens a data channel. If the application needs to transform data as it is being sent or received, and cannot use the **GetFile** or **PutFile** methods, then use the **OpenFile** method to open a data channel with the server.

By default, file names which are sent to the server using the **Command** method are sent as ANSI characters. If the server supports UTF-8 encoded file names, the **Encoding** property can be used to specify that file names with non-ASCII characters should be sent as UTF-8 encoded values. It is important to note that this option is only available if the server advertises support for UTF-8 and permits that encoding type.

See Also

Encoding Property, Features Property, ResultCode Property, ResultString Property, OnCommand Event

Connect Method

Establish a connection with a server.

Syntax

object.**Connect**([*RemoteHost*], [*RemotePort*], [*UserName*], [*Password*], [*Account*], [*Timeout*], [*Options*])

Parameters

RemoteHost

A string which specifies the host name or IP address of the server. If this argument is not specified, it defaults to the value of the **HostAddress** property if it is defined. Otherwise, it defaults to the value of the **HostName** property.

RemotePort

A number which specifies the port to connect to on the server. If this argument is not specified, it defaults to the value of the **RemotePort** property. A value of zero indicates that the default port number for this service should be used to establish the connection. If the secure port number is specified, an implicit TLS connection will be established by default.

UserName

A string that specifies the user name to be used to authenticate the current client session. If this argument is omitted, then the value of the **UserName** property will be used. If the user name is specified as an empty string, then the login is considered to be anonymous.

Password

A string that specifies the password to be used to authenticate the current client session. If this argument is omitted, then the value of the **Password** property will be used. This argument may be empty string if no password is required for the specified user, or if no username has been specified. If you are connecting to a server using SFTP (SSH) and the server requires public/private key authentication, set the **PrivateKey** property to the full path of the private key file prior to calling this method.

Account

A string that specifies the account name to be used to authenticate the current client session. If this argument is omitted, then the value of the **Account** property will be used. This parameter may be an empty string if no account name is required for the specified user.

Timeout

The number of seconds that the client will wait for a response before failing the operation. If this argument is not specified, the value of the **Timeout** property will be used as the default.

Options

A numeric value which specifies one or more options. If this argument is omitted or a value of zero is specified, a default, standard connection will be established. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description
ftpOptionPassive	This option specifies the client should attempt to establish a passive connection to the server. This means that

	instead of the client opening a port on the local system and waiting for the server to establish a connection back to the client, the client will establish a second data connection to the server. This mode is recommended for most systems that are behind a NAT router or firewall.
ftpOptionFirewall	This option specifies the client should always use the host IP address to establish the data connection with the server, not the address returned by the server in response to the PASV command. This option may be necessary if the server is behind a router that performs Network Address Translation (NAT) and it returns an unreachable IP address for the data connection. If this option is specified, it will also enable passive mode data transfers.
ftpOptionNoAuth	This option specifies the server does not require authentication, or that it requires an alternate authentication method. When this option is used, the client connection is flagged as authenticated as soon as the connection to the server has been established. Note that using this option to bypass authentication may result in subsequent errors when attempting to retrieve a directory listing or transfer a file. It is recommended that you consult the technical reference documentation for the server to determine its specific authentication requirements.
ftpOptionKeepAlive	This option specifies the client should attempt to keep the

	connection with the server active for an extended period of time. It is important to note that regardless of this option, the server may still choose to disconnect client sessions that are holding the command channel open but are not performing file transfers.	
&H200	ftpOptionVirtualHost	This option specifies the server supports virtual hosting, where multiple domains are hosted by a server using the same external IP address. If this option is enabled, the client will send the HOST command to the server upon establishing a connection.
&H400	ftpOptionVerify	This option specifies that file transfers should be automatically verified after the transfer has completed. If the server supports the XMD5 command, the transfer will be verified by calculating an MD5 hash of the file contents. If the server does not support the XMD5 command, but does support the XCRC command, the transfer will be verified by calculating a CRC32 checksum of the file contents. If neither the XMD5 or XCRC commands are supported, the transfer is verified by comparing the size of the file. Automatic file verification is only performed for binary mode transfers because of the end-of-line conversion that may occur when text files are uploaded or downloaded.
&H800	ftpOptionTrustedSite	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 the TLS protocol.
&1000	ftpOptionSecure	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure

		connections using the TLS protocol.
&H2000	ftpOptionSecureExplicit	This option specifies the client should use the AUTH TLS-P command to negotiate an explicit secure connection. Some servers may only require this when connecting to the server on ports other than 990.
&H4000	ftpOptionSecureShell	This option specifies the client should use the Secure Shell (SSH) protocol to establish the connection. This option will automatically be selected if the connection is established using port 22, the default port for SSH, and is only required if the server is configured to use a non-standard port number.
&H8000	ftpOptionSecureFallback	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.
&H10000	ftpOptionTunnel	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established. This option also forces all connections to be outbound and enables the firewall compatibility features in the client.
&H20000	ftpOptionKeepAliveData	This option specifies the client should attempt to keep the control connection active during a file transfer. Normally, when a data transfer is in progress, no additional commands are issued on the control channel until the transfer completes. Specifying this option automatically enables the **ftpOptionKeepAlive** option and forces the client to continue to issue NOOP commands during the file transfer. This option

		only applies to FTP and FTPS connections and has no effect on connections using SFTP (SSH).
&40000	ftpOptionPreferIPv6	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.
&100000	ftpOptionHiResTimer	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller files being uploaded or downloaded using fast network connections.
&200000	ftpOptionTLSReuse	This option specifies that TLS session reuse should be enabled for secure data connections. Some servers may require this option be enabled, although it should only used when required. This option is only valid for secure FTP (FTPS) connections and is not used with SFTP or secure HTTP connections. See the remarks below for more information.

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

If the *Options* argument is omitted, then the value of the **Options** property will be used. Note that specifying the **ftpOptionPassive** option is the same as setting the **Passive** property to True.

If the **ftpOptionNoAuth** option is specified, the control will not attempt to authenticate the client session. Note that you may still explicitly call the **Login** method after the connection has been established. This option cannot be used with SFTP (SSH) connections because client authentication is always performed as part of the connection process.

If the **ftpOptionSecureExplicit** option is specified, the client will first send an AUTH TLS command to the server. If the server does not accept this command, it will then send an AUTH SSL command. If both commands are rejected by the server, an explicit TLS session cannot be established. By default,

both the command and data channels will be encrypted when a secure connection is established. To change this, set the **ChannelMode** property.

The **ftpOptionTLSReuse** option is only supported on Windows 8.1 or Windows Server 2012 R2 and later platforms. This option is not compatible with legacy servers built using OpenSSL 1.0.2 and earlier versions which do not provide Extended Master Secret (EMS) support as outlined in RFC7627. This option will be ignored on Windows 7 platforms because they do not support TLS session reuse functionality.

Example

```
' Establish a secure TLS connection on port 21
FtpClient1.HostName = "file.server.tld"
FtpClienn1.RemotePort = 21
FtpClient1.UserName = "username"
FtpClient1.Password = "secret"
FtpClient1.Options = ftpOptionPassive Or ftpOptionSecureExplicit or
ftpOptionTLSReuse

nError = FtpClient1.Connect()
If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

ActivePort Property, HostAddress Property, HostName Property, Options Property, Passive Property. Password Property. PrivateKey Property. RemotePort Property, URL Property, UserName Property, Disconnect Method, OnConnect Event

CreateFile Method

Create a new file or overwrite an existing file.

Syntax

object.**CreateFile**(*RemoteFile*)

Parameters

RemoteFile

A string which specifies the name of the file to create on the server. The file pathing and name conventions must be that of the server.

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 **CreateFile** method creates a new file on the server using the specified file name. The **Write** method may then be used to copy data to the open file. The user must have the appropriate permission to create the file on the server.

See Also

CloseFile Method, OpenFile Method, Write Method

DeleteFile Method

Delete a file on the server.

Syntax

object.**DeleteFile**(*RemoteFile*)

Parameters

RemoteFile

A string which specifies the name of the file on the server that will be deleted. The file pathing and name conventions must be that of the server.

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 **DeleteFile** method deletes an existing file from the server. The user must have the appropriate permission to delete the specified file.

See Also

RenameFile Method

Disconnect Method

Terminate the connection with a server.

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

IsConnected Property, Connect Method, OnDisconnect Event

GetData Method

Download the contents of a file on the server and return it in a string or byte array.

Syntax

object.**GetData**(*RemoteFile*, *Buffer*, [*Length*], [*Reserved*])

Parameters

RemoteFile

A string that specifies the file on the server that will be transferred to the local system. The file pathing and name conventions must be that of the server.

Buffer

This parameter specifies the local buffer that the data will be stored in. If the variable is a **String** type, then the data will be returned as a string of characters. This is the most appropriate data type to use if the file on the server is a text file. If the remote file contains binary data, it is recommended that a **Byte** array variable be specified as the argument to this method.

Length

An optional integer argument passed by reference which will specify the amount of data received from the server when the method returns. For strings, this value specifies the number of characters that were returned by the server. For byte arrays, this value specifies the number of bytes that were returned.

Reserved

An argument reserved for future expansion. This argument should always be omitted or specified as a numeric value of zero.

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 **GetData** method transfers data from a file on the server to the local system, storing it in the specified buffer . This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

If you are returning the data into a **String** variable and the text you receive appears to corrupted or characters are being replaced with question marks or other symbols, it is likely the file on the server is using a different character encoding. Most applications use UTF-8 encoding to represent non-ASCII characters; however, some text files may use a localized character set rather than using Unicode. Using the **GetText** and **PutText** methods in combination with this property will change how that text is converted to Unicode.

The value returned in the ***Length*** parameter may not be identical to the size of a text file on the server. The end-of-line conventions may differ between the server and the local system, and conversion to Unicode may cause differences in the character count. For example, if this method is used to download a UTF-8 encoded text file which includes non-ASCII characters, those characters will be converted to 16-bit Unicode characters. If you want an exact copy of the file as it is on the server, make sure the **FileType** property is set to **ftpFileTypeBinary** and store the data in a **Byte** array instead of a **String** variable.

See Also

CodePage Property, FileType Property, GetFile Method, GetText Method, PutData Method, PutFile Method, PutText Method, OnProgress Event

GetDirectory Method

Return the current working directory on the server.

Syntax

object.**GetDirectory**(*RemotePath*)

Parameters

RemotePath

A string variable which will contain the current working directory when the method returns.

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 **GetDirectory** method returns the current working directory on the server. The method sends the PWD command to the server.

See Also

ChangeDirectory, CloseDirectory Method, OpenDirectory Method, ReadDirectory Method

GetFile Method

Copy a file from the server to the local system.

Syntax

object.**GetFile**(*LocalFile*, *RemoteFile*, [*Options*], [*Offset*])

Parameters

LocalFile

A string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

RemoteFile

A string that specifies the file on the server that will be transferred to the local system. The file pathing and name conventions must be that of the server.

Options

A numeric bitmask which specifies one or more options. This argument may be any one of the following values:

Value	Description
ftpTransferDefault	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the remote file. If the Options argument is omitted, this is the transfer mode which will be used.
ftpTransferAppend	This option specifies that if the local file exists, the contents of file on the server is appended to the local file. If the local file does not exist, it is created.

Offset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

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 **GetFile** method copies an existing file from the server to the local system. This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

BufferSize Property, Priority Property, GetData Method, GetMultipleFiles Method, PutData Method, PutFile Method, VerifyFile Method, OnGetFile Event, OnProgress Event

GetFileList Method

Return an unparsed list of files in the specified directory.

Syntax

object.**GetFileList**(*RemotePath*, *Buffer*, [*Length*], [*Options*])

Parameters

RemotePath

A string which specifies the name of a directory on the server. The list of files and subdirectories in that directory will be returned to the client. To obtain a list of files in the current working directory on the server, use an empty string.

Buffer

A buffer that the data will be stored in. It is recommend that a **String** variable type is used, although it is also possible to provide a **Byte** array as this argument, in which case the file listing will converted to ANSI characters and fill the array. Any other variable type will cause this method to throw an exception.

Length

A numeric value which specifies the maximum number of characters to read. If the argument is omitted, then the maximum size of the buffer will be calculated automatically. In most cases, it is not necessary to provide this argument.

Options

A numeric value which specifies how the list of files should be returned by the server. It may be one of the following values:

Value	Description
ftpListDefault	This option specifies the server should return a complete file list, providing all of the information available about that file. This typically includes the date and time the file was last modified, the size of the file and access rights. This option is the default, and will be used if the argument is omitted from the method call.
ftpListNameOnly	This option specifies the server should only return a list of file names, with no additional information. This option may be used if the server returns the file listing in a format that is not recognized by the control.

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 **GetFileList** method returns a list of files in the specified directory, copying the data to a string buffer. Unlike the **ReadDirectory** method that parses a directory listing, this method returns the unparsed file list data. The actual format of the data that is returned depends on the operating system and how the server implements file listings. For example, UNIX servers typically return the output from the **/bin/ls** command.

Some servers may not support file listings for any directory other than the current working directory. If an error is returned when specifying a directory name, try changing the current working directory

using the **ChangeDirectory** method and then call this method again, an empty string as the ***RemotePath*** parameter.

This method can be particularly useful when the client is connected to a server that returns file listings in a format that is not recognized by the control. The application can retrieve the unparsed file listing from the server and parse the contents. Note that if you specify the **ftpListNameOnly** option, the data will only contain a list of file names and there will be no way for the application to know if they represent a regular file or a subdirectory.

This method is supported for both FTP and SFTP (SSH) connections, however the format of the data may differ depending on which protocol is used. Most UNIX based FTP servers will not list files and subdirectories that begin with a period, however most SFTP servers will return a list of all files, even those that begin with a period.

This method will cause the current thread to block until the file listing completes, a timeout occurs or the operation is canceled.

See Also

ChangeDirectory Method, OpenDirectory Method, ReadDirectory Method

GetFilePermissions Method

Return the access permissions for a file on the server.

Syntax

object.**GetFilePermissions**(*RemoteFile*, *FilePerms*)

Parameters

RemoteFile

A string that specifies the name of the file that the access permissions are to be returned for. The filename cannot contain any wildcard characters.

FilePerms

A numeric variable which is set to the file permissions when the method returns. The file permissions are represented as bit flags, and may be one or more of the following values:

Value	Description
ftpPermWorldExecute	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.
ftpPermWorldWrite	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
ftpPermWorldRead	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
ftpPermGroupExecute	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermGroupWrite	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermGroupRead	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
ftpPermOwnerExecute	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermOwnerWrite	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this

	grants the user the right to create and delete files.
ftpPermOwnerRead	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
ftpPermSymbolicLink	The file is a symbolic link to another file. Symbolic links are special types of files found on UNIX based systems which are similar to Windows shortcuts.

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 **GetFilePermissions** method returns information about the access permissions for a specific file on the server. This method uses the STAT command to retrieve information about the specified file. If the server does not support the use of this command, an error will be returned. You can use the **Features** property to determine what features are available and/or enabled on the server.

Note that on some systems, the STAT command will not return information on files that contain spaces or tabs in the filename. In this case, the method will fail.

Example

The following example demonstrates how to retrieve the access permissions for a file and then test to see if the file can be read by the owner of that file:

```
nError = FtpClient1.GetFilePermissions(strFileName, nFilePerms)
If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

If (nFilePerms And ftpPermOwnerRead) <> 0 Then
    MsgBox "The file " & strFileName & " can be read by the owner"
End If
```

See Also

Features Property, SetFilePermissions Method

GetFileSize Method

Returns the size of the specified file on the server.

Syntax

object.**GetFileSize**(*RemoteFile*, *FileSize*)

Parameters

RemoteFile

A string that specifies the name of the file on the server. The filename cannot contain any wildcard characters and must follow the naming conventions of the operating system the server is hosted on.

FileSize

A numeric variable which will be set to the size of the file on the server. Note that if the variable is not large enough to contain the file size, an overflow error will occur. This parameter must be passed by reference.

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

This method uses the SIZE command to determine the length of the specified file. Not all servers implement this command, in which case the method will fail. You can use the **Features** property to determine what features are available and/or enabled on the server.

Note that if the file on the server is a text file, it is possible that the value returned by this method will not match the size of the file when it is downloaded to the local system. This is because different operating systems use different sequences of characters to mark the end of a line of text, and when a file is transferred in text mode, the end of line character sequence is automatically converted to a carriage return-linefeed, which is the convention used by the Windows platform.

Some FTP servers will refuse to return the size of a file if the current file type is set to **ftpFileTypeText** because the size of a text file on the server may not accurately reflect what the size of the file will be on the local system.

Example

The following example demonstrates how to retrieve the size a file on the server:

```
Dim nFileSize As Long

nError = FtpClient1.GetFileSize(strFileName, nFileSize)
If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

MsgBox "The size of " & strFileName & " is " & nFileSize " bytes"
```

See Also

Features Property, GetFileStatus Method, GetFileTime Method

GetFileStatus Method

Return status information about a specific file.

Syntax

object.**GetFileStatus**(*FileName*, [*FileLength*], [*FileDate*], [*FileOwner*], [*FileGroup*], [*FilePerms*], [*IsDirectory*])

Parameters

FileName

A string which specifies the name of the file that status information will be returned for.

FileLength

An optional numeric argument which will specify the size of the file on the server. Note that if this is a text file, the file size may be different on the server than it is on the local system. This is because different operating systems use different conventions that indicate the end of a line and/or the end of the file. On MS-DOS and Windows platforms, directories have a file size of zero bytes. This parameter must be passed by reference.

FileDate

An optional string argument which will specify the date and time the file was created or last modified on the server. The date format that is returned is expressed in local time (in other words, the timezone of the server is not taken into account) and depends on both the local host settings via the Control Panel and the format of the date and time information returned by the server. This parameter must be passed by reference.

FileOwner

An optional string argument which will specify the owner of the file on the server. On some platforms, this information may not be available for security reasons if an anonymous login session was specified. This parameter must be passed by reference.

FileGroup

An optional string argument which will specify the group that the file owner belongs to. On some platforms, this information may not be available for security reasons if an anonymous login session was specified. This parameter must be passed by reference.

FilePerms

An optional numeric argument which will specify the permissions assigned to the file. This value is actually a combination of one or more bit flags that specify the individual permissions for the file owner, group and world (all other users). This parameter must be passed by reference. The permissions are as follows:

Value	Description
ftpPermWorldExecute	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.
ftpPermWorldWrite	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
ftpPermWorldRead	All users have permission to open the file for reading. This

	permission grants any user the right to download the file to the local system.
ftpPermGroupExecute	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermGroupWrite	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermGroupRead	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
ftpPermOwnerExecute	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermOwnerWrite	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermOwnerRead	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
ftpPermSymbolicLink	The file is a symbolic link to another file. Symbolic links are special types of files found on UNIX based systems which are similar to Windows shortcuts.

IsDirectory

An optional boolean value which will specify if the file is a directory or a regular file. This parameter must be passed by reference.

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 **GetFileStatus** method returns information about the specified file. The filename must be specified using the server file naming conventions, and cannot include wildcard characters. The primary difference between using this method and using the **OpenDirectory** and **ReadDirectory** methods to obtain file information is that the file status information is returned on the command channel. This method cannot be used while a file transfer is in progress or while a file listing is being returned by the server.

Note that this method requires that the server return file status information in response to the STAT command. Some servers, for example on VMS platforms, do not provide this information. On some

systems, the STAT command will not return information on files that contain spaces or tabs (whitespace) in the filename. In this case, the method will set the specified arguments to empty strings and zero values.

See Also

ParseList Property, CloseDirectory Method, OpenDirectory Method, ReadDirectory Method

GetFileTime Method

Returns the modification date and time for specified file on the server.

Syntax

object.**GetFileTime**(*RemoteFile*, *FileDate*)

Parameters

RemoteFile

A string that specifies the name of the file on the server. The filename cannot contain any wildcard characters and must follow the naming conventions of the operating system the server is hosted on.

FileDate

A variable that will be set to the date and time that the file was last modified. The variable's data type must either be **Variant**, **String** or **Date**. This parameter must be passed by reference.

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 **GetFileTime** method returns the modification date and time for the specified file on the server using the MTDM command. If the server does not support this command, the method will attempt to use the STAT command to determine the file modification time. You can use the **Features** property to determine what features are available and/or enabled on the server.

The **Localize** property will determine if the returned file time is adjusted for the local timezone.

Example

The following example demonstrates how to retrieve the size a file on the server:

```
Dim dateFileTime As Date

nError = FtpClient1.GetFileTime(strFileName, dateFileTime)
If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

MsgBox strFileName & " was modified on " & dateFileTime
```

See Also

Features Property, Localize Property, GetFileStatus Method, GetFileSize Method, SetFileTime Method

GetFileType Method

Returns the file type for a file on the local system.

Syntax

object.**GetFileType**(*FileName*, *ScanFile*)

Parameters

FileName

A string which specifies the name of a file on the local system. The file does not need to exist if the *ScanFile* parameter is **False**. If an existing file is specified, it cannot be the name of a device or a directory, otherwise the method will fail.

ScanFile

An optional Boolean value which specifies if the contents of the file should be scanned. A value of **False** indicates that only the file extension should be used to determine the file type, while a value of **True** specifies the contents of the file should be examined if the file type cannot be determined based on its extension. If this parameter is omitted, the default value is **False**.

Return Value

An integer value of zero or greater which identifies the file type using the same values as the **FileType** property. If the method fails, it will return -1 indicating an error condition. The value of the **LastError** property can be used to determine the cause of the failure.

Remarks

This method is used to determine the file transfer type to be used when uploading or downloading files. This method is used internally when **ftpFileTypeAuto** is specified as the default file type. The return value may be one of the following:

Value	Description
ftpFileTypeASCII	The file is a text file using the ASCII character set. For those servers which mark the end of a line with characters other than a carriage return and linefeed, it will be converted to the native client format. This is the file type used for directory listings. The constant **ftpFileTypeText** is an alias for this value.
ftpFileTypeEBCDIC	The file is a text file using the EBCDIC character set. Local files will be converted to EBCDIC when sent to the server. Remote files will be converted to the native ASCII character set when retrieved from the server. Not all servers support this file type. It is recommended that you only specify this type if you know that it is required by the server to transfer data correctly.
ftpFileTypeImage	The file is a binary file and no data conversion of any type is performed on the file. This is the default file type for most data files and executable programs. If the type of file cannot be automatically determined, it will always be considered a binary file. If this file type is specified when uploading or downloading text files, the native end-of-line character sequences will be preserved. The constant **ftpFileTypeBinary** is an alias for this value.

If the file extension or contents are not recognized, the default file transfer type for the client session

will be returned. This will usually be **ftpFileTypeImage**, however this can be changed by calling the **AddFileType** method. The file type for the current client session can be explicitly set using the **FileType** property.

If the ***ScanFile*** parameter is True, the local file will be opened in a shared reading mode and up to 4,096 bytes will be examined to determine if it contains binary data. If the file is currently locked or has been opened exclusively by another process, the file type associated with the file extension will be returned instead. Text files which contain UTF-16 text will always return a file type of **ftpFileTypeImage** because they can contain non-ASCII characters and/or embedded null characters.

If the ***ScanFile*** parameter is True and the file type cannot be determined based on the file name extension, the file specified by ***FileName*** must exist and be a regular file. If the file does not exist, an error will be returned and the last error code will be set to **stErrorFileNotFound**. If the ***ScanFile*** parameter is False, no errors will be returned if the file does not exist, the function will only check the file name extension to determine the file type. When downloading a file, the ***ScanFile*** parameter should normally be zero because the local file may not exist yet.

See Also

FileType Property, AddFileType Method, GetFile Method, PutFile Method

GetMultipleFiles Method

Transfer multiple files from the server to the local system.

Syntax

object.**GetMultipleFiles**(*LocalPath*, *RemotePath*, [*FileMask*], [*Reserved*])

Parameters

LocalPath

A string argument which specifies the name of the directory on the local system where the files will be stored. If a file by the same name already exists, it will be overwritten.

RemotePath

A string argument which specifies the name of the directory on the server where the files will be copied from. You must have permission to read the contents of the directory.

FileMask

An optional string argument which specifies the wildcard mask to be used when selecting what files should be transferred. If this argument is omitted, the value of the **FileMask** property will be used. The default value of an empty string indicates that all files in the specified directory should be downloaded. Typically, this argument is a wildcard mask that limits the files downloaded from the server to those which match a specific extension. For example, to download only those files that end in a ".dat" extension, the argument could be specified as "*.dat"

Reserved

An argument reserved for future expansion. This argument should always be omitted or specified as a numeric value of zero.

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 **GetMultipleFiles** method copies multiple files from the server to the local system. If the local file already exists, it is overwritten. This method will cause the current thread to block until all of the files have been transferred, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

BufferSize Property, GetData Method, GetFile Method, PutData Method, PutFile Method, PutMultipleFiles Method, OnGetFile Event, OnProgress Event

GetText Method

Download a text file from the server and store it in string.

Syntax

object.**GetText**(*RemoteFile*, *Buffer*)

Parameters

RemoteFile

A string that specifies the name of a file on the server that will be downloaded. The file pathing and name conventions must be that of the server.

Buffer

This parameter is passed by reference and specifies the string buffer which will contain the text returned by the server. This parameter must be a String or Variant type which will reference a string when the method returns. This method will not accept a byte array as an argument.

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 **GetText** method is used to download the contents of a text file and store it in a String variable. This method should only be used with text files which are known to be textual. For example, it is safe to use this method when downloading an HTML or XML document, but should not be used to download executable or compressed files (such as Microsoft Word documents or Excel spreadsheets) . Always use the **GetData** method if you wish to retrieve binary data and store it in a byte array.

The text document returned by the server is automatically converted to Unicode using the code page specified by the **CodePage** property. Most text files today will use either ASCII or UTF-8 encoding, however some documents may contain text specific to the locale they were created in. Because ASCII is a subset of UTF-8, it is safe to specify UTF-8 encoding for ASCII text documents. If you specify an incorrect code page, this can result in a conversion error.

This method will always attempt to normalize the end-of-line character sequence to use a carriage-return and linefeed (CRLF) pair. This can potentially result in a discrepancy between the size of a text file on the server and the actual length of the string buffer.

This method will always use an ASCII file transfer mode, regardless of the value of the **FileType** property. If the remote file contains binary data, the string buffer may be empty or contain unprintable characters as the result of attempting to convert the data to Unicode.

See Also

CodePage Property, FileType Property, GetData Method, PutData Method, PutText Method, OnProgress Event

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 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 ftpClient = CreateObject("SocketTools.FtpClient.11")

nError = ftpClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize Method

Login Method

Login to the server.

Syntax

object.**Login**([*UserName*], [*Password*], [*Account*])

Parameters

UserName

A string that specifies the name of the user logging into the server. This argument is optional, and if it is omitted, the value of the **UserName** property will be used. If the **UserName** property has not been set, an anonymous user session is established.

Password

A string that specifies the password used to authenticate the user. This argument is optional, and if it is omitted, the value of the **Password** property will be used. If no user name has been specified, then an anonymous user session is established; in this case, the common convention that is used is that the password is specified as the current user's email address.

Account

A string that specifies the account name to be used when authenticating the user. This argument is optional, and if omitted, the value of the **Account** property will be used. An account name should only be specified if required by the server. Most UNIX and Windows based FTP servers do not require an account name.

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 **Login** method identifies the user to the server. If the user name or password is invalid, an error will occur. By default, when a connection is established, the **UserName**, **Password** and **Account** properties are used to automatically log the user in to the server.

This method should only be used after calling the **Logout** method, enabling you to log in as another user during the same session. Not all servers will permit a client to change user credentials during the same session. In most cases, it is preferable to disconnect from the server and re-connect using the new credentials rather than using this method.

This method is not supported with secure connections using the SSH protocol.

See Also

Account Property, Password Property, UserName Property, Connect Method, Logout Method

Logout Method

Log the current user off the server.

Syntax

object.**Logout**

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 **Logout** method logs the current user off the server. The **Login** method may then be used to login as another user during the same session. Note that this method will not terminate the connection with the server.

This method is not supported with secure connections using the SSH protocol.

See Also

Connect Method, Login Method

MakeDirectory Method

Create a new directory on the server.

Syntax

object.**MakeDirectory**(*RemotePath*)

Parameters

RemotePath

A string that specifies the name of the directory to create on the server. The naming and pathing conventions used for the directory must be compatible with what is used on the operating system that hosts the server.

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 **MakeDirectory** method creates a new directory on the server. Note that you must have the appropriate permission to create a directory or an error will occur.

Servers may not support creating multiple subdirectories in a single call, so applications should not assume that this can be done. For example, an error may be returned by the server if the new directory name "/Projects/Today" is specified, but the "/Projects" directory does not already exist.

It is also important to note that files and directories on UNIX based systems are case sensitive, so the directory names "Projects" and "projects" refer to two different directories. This is not the case on Windows systems, where either name would refer to the same directory.

See Also

ChangeDirectory Method, RemoveDirectory Method

OpenDirectory Method

Open the specified directory on the server for reading.

Syntax

object.**OpenDirectory**(*RemotePath*)

Parameters

RemotePath

A string that specifies the name of the directory to open on the server. The naming and pathing conventions used for the directory must be compatible with what is used on the operating system that hosts the server.

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 **OpenDirectory** method opens the specified directory on the server so that the list of files in that directory may be read using the **ReadDirectory** method.

Once all of the files in the directory have been read, the application must call the **CloseDirectory** method in order to close the data channel to the server. Failure to do this will result in an error the next time the application attempts to transfer a file or open another directory.

Note that files and directories on UNIX based systems are case sensitive, so the directory names "Projects" and "projects" refer to two different directories. This is not the case on Windows systems, where either name would refer to the same directory.

See Also

CloseDirectory Method, GetDirectory Method, ReadDirectory Method

OpenFile Method

Open an existing file or creates a new file on the server.

Syntax

object.**OpenFile**(*RemoteFile*, [*FileMode*], [*Offset*])

Parameters

RemoteFile

A string that specifies the name of the file on the server. The filename cannot contain any wildcard characters and must follow the naming conventions of the operating system the server is hosted on.

FileMode

A numeric value which specifies how the file will be accessed. It may be one of the following values:

Value	Description
ftpFileRead	The file is opened for reading on the server. A data channel is created and the contents of the file are returned to the client.
ftpFileWrite	The file is opened for writing on the server. If the file does not exist, it will be created. If it does exist, it will be overwritten.
ftpFileAppend	The file is opened for writing on the server. All data will be appended to the end of the file.

Offset

An optional byte offset which specifies where the file transfer should begin. If this argument is omitted, this specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset using FTP requires that the server support the REST command to restart transfers.

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 **OpenFile** method opens an existing file or creates a file on the server using the specified file name. The **Read** method may then be used to read data from the file and the **Write** method may be used to write data to the file. Once the all of the data has been read or written, the **CloseFile** method must be called to close the data channel.

Only one file may be opened at a time for each client session. Attempting to perform an action such as uploading or downloading another file while a file is currently open will result in an error. Typically this indicates that the application failed to call the **CloseFile** method.

It is strongly recommended that most applications use the **GetFile** or **PutFile** methods to perform file transfers. These methods are easier to use, and have internal optimizations that improves the overall data transfer rate when compared to implementing the file transfer code in your own application.

See Also

CloseFile Method, CreateFile Method, GetFile Method, PutFile Method, Read Method, Write Method

PutData Method

Upload the contents of a string or byte array and store it in a file on the server.

Syntax

object.**PutData**(*RemoteFile*, *Buffer*, [*Length*], [*Reserved*])

Parameters

RemoteFile

A string that specifies the file on the server that will contain the data being transferred. If the file already exists, it will be overwritten. The file pathing and name conventions must be that of the server.

Buffer

This parameter specifies the local buffer that the data will be copied from. If the parameter is a **String** type, then the data will be written as a string of characters. For binary data, it is recommended that this parameter specify a **Byte** array.

Length

An optional integer argument that specifies the amount of data to be copied from the buffer. If this argument is omitted, the entire contents of the buffer is transferred to the server. For strings, this value specifies the number of characters to be copied. For byte arrays, this value specifies the number of bytes.

Reserved

An argument reserved for future expansion. This argument should always be omitted or specified as a numeric value of zero.

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 **PutData** method transfers data from a local buffer and stores it on a file on the server. This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

If you are using a **String** buffer, the contents of the buffer will automatically be converted to UTF-8 encoded text. Most applications support UTF-8 encoding, however if you need to store the text using a different encoding you can change the value of the **CodePage** property and use the **PutText** method instead.

Never use a **String** variable to upload binary data. This method will automatically attempt to convert the contents of the string to UTF-8 encoded text and this can corrupt the data. If you need to upload binary data using the **PutData** method, you should always use a **Byte** array.

The value specified with the ***Length*** parameter may not be identical to the size of the text file created on the server. The end-of-line conventions may differ between the server and the local system, and conversion to Unicode may cause differences in the character count. For example, if this method is used to upload text which includes non-ASCII characters, those characters will be UTF-8 encoded. If you want to upload an exact copy of the data in your buffer, make sure the **FileType** property is set to **ftpFileTypeBinary** and use a **Byte** array instead of a **String** variable.

See Also

CodePage Property, FileType Property, GetData Method, GetFile Method, GetText Method, PutFile Method, PutText Method, OnProgress Event

PutFile Method

Copy a file from the local system to the server.

Syntax

object.**PutFile**(*LocalFile*, *RemoteFile*, [*Options*], [*Offset*])

Parameters

LocalFile

A string that specifies the file on the local system that will be transferred to the server. The file pathing and name conventions must be that of the local host.

RemoteFile

A string that specifies the file on the server that will be created, overwritten or appended to. The file pathing and name conventions must be that of the server.

Options

A numeric bitmask which specifies one or more options. This argument may be any one of the following values:

Value	Description
ftpTransferDefault	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the local file. If the Options argument is omitted, this is the transfer mode which will be used.
ftpTransferAppend	This option specifies that if the remote file exists, the contents of file on the local system is appended to the remote file. If the remote file does not exist, it is created.

Offset

A byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully. Note that specifying a non-zero offset requires that the server support the REST command to restart transfers.

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 **PutFile** method copies an existing file from the local system to the server. This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

BufferSize Property, Priority Property, GetData Method, GetFile Method, PutData Method, PutMultipleFiles Method, VerifyFile Method, OnProgress Event, OnPutFile Event

PutMultipleFiles Method

Transfer multiple files from the local system to the server.

Syntax

object.**PutMultipleFiles**(*LocalPath*, *RemotePath*, [*FileMask*], [*Reserved*])

Parameters

LocalPath

A string argument which specifies the name of the directory on the local system where the files will be copied from. You must have permission to read the contents of the directory.

RemotePath

A string argument which specifies the name of the directory on the server where the files will be stored. You must have permission to modify the contents of the directory and create files.

FileMask

An optional string argument which specifies the wildcard mask to be used when selecting what files should be transferred. If this argument is omitted, the value of the **FileMask** property will be used. The default value of an empty string indicates that all files in the specified directory should be uploaded. Typically, this argument is a wildcard mask that limits the files uploaded to the server to those which match a specific extension. For example, to upload only those files that end in a ".dat" extension, the argument could be specified as "*.dat"

Reserved

An argument reserved for future expansion. This argument should always be omitted or specified as a numeric value of zero.

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 **PutMultipleFiles** method copies multiple files from the local system to the server. If the remote file already exists, it is overwritten. This method will cause the current thread to block until all of the files have been transferred, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

BufferSize Property, GetData Method, GetFile Method, GetMultipleFiles Method, PutData Method, PutFile Method, OnProgress Event, OnPutFile Event

PutText Method

Upload the contents of a string buffer and store it in a text file on the server.

Syntax

object.**PutText**(*RemoteFile*, *Buffer*)

Parameters

RemoteFile

A string that specifies the name of a file on the server that will be downloaded. The file pathing and name conventions must be that of the server.

Buffer

A string which contains the text to be stored on the server. This method will not accept a Byte array as an argument.

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 **PutText** method is used to upload the contents of a string and store it as a text file on the server. Although a String variable may contain binary data, this method should only be used with strings which contain printable text. Always use the **PutData** method if you wish to upload binary data, using a Byte array instead of a String variable.

The text uploaded to the server is automatically converted from Unicode using the code page specified by the **CodePage** property. By default, text will be automatically converted to use UTF-8 encoding, however you can change this if you prefer to store the file using a different localized encoding. In most cases it is recommended you use UTF-8 to ensure the broadest compatibility with other applications.

This method will always attempt to normalize the end-of-line character sequence to match what is used on the server. This can potentially result in a discrepancy between the size of a text file on the server and the actual length of the string buffer. For example, Windows uses a carriage return and linefeed pair (CRLF) to indicate the end of a line of text. If you are storing the text in a file on a UNIX system, it will be changed to use only a linefeed (LF) to indicate the end of a line.

This method will always use an ASCII file transfer mode, regardless of the value of the **FileType** property. If the string buffer contains binary data, the resulting file may be empty or contain unprintable characters as the result of the Unicode text conversion.

See Also

CodePage Property, FileType Property, GetData Method, GetText Method, PutData Method, OnProgress Event

Read Method

Return data read from the server.

Syntax

object.**Read**(*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

The number of bytes actually read from the server 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 server, 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 UTF-8 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

IsConnected Property, IsReadable Property, Write Method, OnRead Event, OnWrite Event

ReadDirectory Method

Read a directory entry from the server.

Syntax

object.**ReadDirectory**(*FileName*, [*FileLength*], [*FileDate*], [*FileOwner*], [*FileGroup*], [*FilePerms*], [*IsDirectory*])

Parameters

FileName

A string which will specify the name of the file that status information will be returned for.

FileLength

An optional numeric argument which will specify the size of the file on the server. Note that if this is a text file, the file size may be different on the server than it is on the local system. This is because different operating systems use different conventions that indicate the end of a line and/or the end of the file. On MS-DOS and Windows platforms, directories have a file size of zero bytes. This parameter must be passed by reference.

FileDate

An optional string argument which will specify the date and time the file was created or last modified on the server. The date format that is returned is expressed in local time (in other words, the timezone of the server is not taken into account) and depends on both the local host settings via the Control Panel and the format of the date and time information returned by the server. This parameter must be passed by reference.

FileOwner

An optional string argument which will specify the owner of the file on the server. On some platforms, this information may not be available for security reasons if an anonymous login session was specified. This parameter must be passed by reference.

FileGroup

An optional string argument which will specify the group that the file owner belongs to. On some platforms, this information may not be available for security reasons if an anonymous login session was specified. This parameter must be passed by reference.

FilePerms

An optional numeric argument which will specify the permissions assigned to the file. This value is actually a combination of one or more bit flags that specify the individual permissions for the file owner, group and world (all other users). This parameter must be passed by reference. The permissions are as follows:

Value	Description
ftpPermWorldExecute	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.
ftpPermWorldWrite	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
ftpPermWorldRead	All users have permission to open the file for reading. This

	permission grants any user the right to download the file to the local system.
ftpPermGroupExecute	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermGroupWrite	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermGroupRead	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
ftpPermOwnerExecute	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermOwnerWrite	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermOwnerRead	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
ftpPermSymbolicLink	The file is a symbolic link to another file. Symbolic links are special types of files found on UNIX based systems which are similar to Windows shortcuts.

For the proprietary Sterling directory formats, the status code is returned in the ***FilePerms*** argument. This value is a combination of bits. Bits 0-25 correspond to letters of the alphabet, most of which have distinct meanings in the Sterling formats.

Letter code	Bit position	Hexadecimal value
A	1h	
B	2h	
C	4h	
n-th letter of alphabet	n-1	2 to the (n-1) power
Z	2000000h	

For the proprietary Sterling directory formats, bits 26-31 represent the transfer protocol associated with the file:

Protocol	Bit position	Hexadecimal value	Value
TCP	4000000h	ftpSterlingStatusTcp	

FTP	8000000h	ftpSterlingStatusFtp
BSC	10000000h	ftpSterlingStatusBsc
ASC	20000000h	ftpSterlingStatusAsc
FTS	40000000h	ftpSterlingStatusFts
other	80000000h	ftpSterlingStatusOther

IsDirectory

An optional boolean value which will specify if the file is a directory or a regular file. This parameter must be passed by reference.

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 **ReadDirectory** method reads the next entry from the directory listing. This method can only be used after the **OpenDirectory** method has been called to begin the transfer of file information to the client.

See Also

CloseDirectory Method, OpenDirectory Method

RemoveDirectory Method

Remove a directory on the server.

Syntax

object.**RemoveDirectory**(*RemotePath*)

Parameters

RemotePath

A string that specifies the name of the directory to remove from the server. The naming and pathing conventions used for the directory must be compatible with what is used on the operating system that hosts the server.

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 **RemoveDirectory** method removes an existing directory on the server. You must have the appropriate permission to remove the directory, or an error will occur. Note that most operating systems will not permit you to remove a directory that contains files or other subdirectories.

See Also

ChangeDirectory Method, MakeDirectory Method

RenameFile Method

Rename or move a file on the server. The original file must exist and the current user must have the appropriate permissions to change the file name.

Syntax

object.**RenameFile**(*OldName*, *NewName*)

Parameters

OldName

A string that specifies the name of the file to be renamed on the server. The file must exist on the server, otherwise an error will be returned.

NewName

A string that specifies the new name for the file on the server. The naming conventions used for the file must be compatible with what is used on the operating system that hosts the server.

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

If the ***NewName*** parameter specifies a file which already exists on the server, the contents of that file will be deleted and replaced by the contents of the file specified by ***OldName***. If you want to prevent this from occurring, your application can check if a file with the new name exists by calling the **GetFileStatus** method. If successful, you can warn the user the rename operation would delete the contents of the new file as a consequence of renaming the original file.

If the connection was established using the standard FTP or FTPS protocols, this method causes two separate commands to be sent to the server, RNFR and RNTO. If either command fails, the method will fail and return an error code. If the connection was established using SFTP (SSH) the client will request the server rename the file atomically by specifying the internal SSH_FXF_RENAME_ATOMIC option. If the server does not support this option and ***NewName*** already exists on the server, this method will attempt to delete the file and retry the rename operation.

This method can be used to move a file from one folder to another on the server, as long as the user has the appropriate permissions required to perform the operation. If the new file name includes a path, that path must already exist on the server or the method will fail.

If the old and new file names specify folders instead of regular files, the server may or may not allow the operation, depending on the access rights for those folders and the server's configuration. Some servers may allow a simple rename operation within the same parent folder, but may not permit you to move the folder to another location in the filesystem.

There is no guarantee the rename operation will be performed in an atomic fashion if multiple client sessions attempt to rename the same file at the same time. For example, it is possible that one client could attempt to rename a file while another client is already in the process of moving the file to a new folder. In this case, the server may respond with an error indicating the file cannot be found, an access denied error or a general failure error.

See Also

DeleteFile Method, GetFile Method, GetFileStatus Method, PutFile 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, open network connections will be closed and any handles allocated by the control will be released.

The **Reset** and **Uninitialize** methods will abort all active background transfers and wait for those tasks to complete before returning to the caller. It is recommended that your application explicitly wait for background transfers to complete or abort them using this method before allowing the program to terminate. This will ensure that your program can perform any necessary cleanup operations. If there are active background tasks running at the time that the control instance is destroyed, it can force the control to stop those worker threads immediately without waiting for them to terminate gracefully.

See Also

Cancel Method, Initialize Method, Uninitialize Method

SetFilePermissions Method

Change the access permissions for a file on the server.

Syntax

object.**SetFilePermissions**(*RemoteFile*, *FilePerms*)

Parameters

RemoteFile

A string that specifies the name of the file that the access permissions are to be returned for. The filename cannot contain any wildcard characters.

FilePerms

A numeric value which specifies the new permissions for the file. The file permissions are represented as bit flags, and may be one or more of the following values:

Value	Description
ftpPermWorldExecute	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.
ftpPermWorldWrite	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
ftpPermWorldRead	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
ftpPermGroupExecute	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermGroupWrite	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermGroupRead	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
ftpPermOwnerExecute	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermOwnerWrite	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right to replace the file. If this permission is set for a directory, this

	grants the user the right to create and delete files.
ftpPermOwnerRead	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.

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 **SetFilePermissions** method uses the SITE CHMOD command to set the permissions for the file. This command is typically only supported on servers that are hosted on UNIX based systems. If the command is not supported, an error will be returned. You can use the **Features** property to determine what features are available and/or enabled on the server.

Users who are familiar with the UNIX operating system will recognize the **chmod** command used to change the file permissions. However, it should be noted that the numeric value used as an argument to the command is in octal, not decimal. For example, issuing the command **chmod 644 filename.txt** on a UNIX based system will make the file readable and writable by the owner, and readable by other users in the owner's group as well as all other users. The value 644 is an octal value, which is equivalent to the decimal value 420. If you were to mistakenly specify 644 as the value for the ***Permissions*** argument, rather than the decimal value of 420, the permissions on the file would be incorrect. It is strongly recommended that you use the pre-defined constants to prevent this sort of error.

Visual Basic allows you to specify an integer value in octal by prefixing it with &O. For example, &O644 could be used as the file permissions value. C and C++ consider any integer with a preceding 0 to be an octal number, so 0644 would be a valid permissions value. Consult the technical reference for your programming language if you are unsure if it supports expressing integer constants in octal.

Example

The following example demonstrates how to change the permissions so that only the owner can read and write to the file:

```
nFilePerms = ftpPermOwnerRead Or ftpPermOwnerWrite
nError = FtpClient1.SetFilePermissions(strFileName, nFilePerms)
If nError > 0 Then
    MsgBox FtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

Features Property, GetFilePermissions Method

SetFileTime Method

Changes the modification date and time for a file on the server.

Syntax

object.**SetFileTime**(*RemoteFile*, *FileTime*)

Parameters

RemoteFile

A string that specifies the name of the file on the server. The filename cannot contain any wildcard characters and must follow the naming conventions of the operating system the server is hosted on.

FileTime

A string that specifies the new date and time for the file. The date must be in a format recognized by the local system, otherwise an error will occur. The date and time value must also be specified in UTC (Coordinated Universal Time), not local time.

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 **SetFileTime** method changes the modification date and time for the specified file on the server. When connected to an FTP server, this method uses the MTDM command to change the modification time for the file. If the server does not support this command, the method will return an error. Note that some servers only support the MDTM command to return, but not change, the file modification time.

See Also

Localize Property, GetFileStatus Method, GetFileSize Method, GetFileTime Method

TaskAbort Method

Abort the specified asynchronous task.

Syntax

object.**TaskAbort** ([*TaskId*], [*Milliseconds*])

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

Milliseconds

An optional integer value that specifies the number of milliseconds to wait for the background task to abort.

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 **TaskAbort** method signals the background worker thread associated with the task ID to abort the current operation and terminate as soon as possible. If the ***TaskId*** parameter is omitted, this method will abort all active background file transfers, otherwise it will only abort the specified task. If the ***Milliseconds*** parameter is omitted or has a value of zero, the method returns immediately after the background thread has been signaled. If the ***Milliseconds*** parameter is non-zero, the method will wait that amount of time for the background thread to terminate.

The **Reset** and **Uninitialize** methods will abort all active background transfers and wait for those tasks to complete before returning to the caller. It is recommended that your application explicitly wait for background transfers to complete or abort them using this method before allowing the program to terminate. This will ensure that your program can perform any necessary cleanup operations. If there are active background tasks running at the time that the control instance is destroyed, it can force the control to stop those worker threads immediately without waiting for them to terminate gracefully.

See Also

TaskCount Property, TaskList Property, TaskDone Method, TaskWait Method

TaskDone Method

Determine if an asynchronous task has completed.

Syntax

object.**TaskDone** ([*TaskId*])

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

Return Value

A Boolean value that specifies if the task has completed. A return value of **True** specifies that the background task has completed. A return value of **False** specifies that the background task is active.

Remarks

The **TaskDone** method is used to determine if the specified asynchronous task has completed. If the *TaskId* parameter is omitted, the method will check the status of the last background task that was started.

If you use this method to poll the status of a background task from within the main UI thread, you must ensure that Windows messages are processed so that the application remains responsive to the end-user. To check if a background transfer has completed, it is recommended that you use a timer to periodically call this method rather than calling it repeatedly within a loop.

To determine if the task completed successfully, the **TaskWait** method will provide the last error code associated with the task. Note that if this method returns **True**, it is guaranteed that calling **TaskWait** using the same task ID will return the error code to the caller immediately without causing the application to block.

See Also

TaskCount Property, TaskId Property, TaskList Property, TaskAbort Method, TaskWait Method

TaskResume Method

Resume execution of an asynchronous task.

Syntax

object.**TaskResume** (*TaskId*)

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

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 **TaskResume** method resumes execution of the background worker thread that was previously suspended using the **TaskSuspend** method. If the ***TaskId*** parameter is omitted, the method will resume execution of the last background task that was started.

See Also

TaskId Property, TaskSuspend Method, TaskWait Method

TaskSuspend Method

Suspend execution of an asynchronous task.

Syntax

object.**TaskSuspend** (*TaskId*)

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

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 **TaskSuspend** method will suspend execution of the background worker thread associated with the task. If the ***TaskId*** parameter is omitted, the method will suspend the last background task that was started.

Once the task has been suspended, it will no longer be scheduled for execution, however the client session will remain active and the task may be resumed using the **TaskResume** method. Note that if a task is suspended for a long period of time, the background operation may fail because it has exceeded the timeout period imposed by the server.

See Also

TaskId Property, TaskResume Method, TaskWait Method

TaskWait Method

Wait for an asynchronous task to complete.

Syntax

object.**TaskWait** ([*TaskId*], [*Milliseconds*], [*TimeElapsed*], [*TaskError*])

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

Milliseconds

An optional integer value that specifies the number of milliseconds to wait for the background task to complete.

TimeElapsed

An optional integer value passed by reference that will contain the elapsed time for the task in milliseconds when the method returns. If this information is not required, this parameter may be omitted. This parameter is ignored if the ***TaskId*** parameter is omitted.

TaskError

An optional integer value passed by reference that will contain the last error code for the task when the method returns. If this information is not required, this parameter may be omitted. This parameter is ignored if the ***TaskId*** parameter is omitted.

Return Value

A Boolean value that specifies if the task has completed. A return value of **True** specifies that the background task has completed. A return value of **False** specifies that the background task is active.

Remarks

The **TaskWait** method waits for the specified task to complete. If the ***TaskId*** parameter is omitted, this method will wait for all active tasks to complete. If a task ID is specified and the ***Milliseconds*** parameter is non-zero, this method will cause the current working thread to block until the task completes or the amount of time exceeds the number of milliseconds specified by the caller. If the ***Milliseconds*** parameter is zero, then this function will poll the status of the task and return immediately to the caller. If the ***Milliseconds*** parameter is omitted, then the method will wait an infinite period of time for the task to complete.

If the specified task has already completed at the time this method is called, the method will return immediately without causing the current thread to block. If the ***TimeElapsed*** parameter has been specified, it will contain the number of milliseconds that it took for the task to complete. If the ***TaskError*** parameter has been specified, it will contain the last error code value that was set by the worker thread before it terminated. If the ***TaskError*** value is zero, that means that the background task was successful and no error occurred. A non-zero value will indicate that the background task has failed.

You should not call this method from the main UI thread with a long timeout period to wait for a background task to complete. Windows messages will not be processed while this method is blocked waiting for the background task to complete, and this can cause your application to appear non-responsive to the end-user. If you have a GUI application and you need to determine if a background task has finished, create a timer to periodically call the **TaskDone** method. When it returns **True** (indicating that the task has completed), you can safely call **TaskWait** to obtain the elapsed time and last error code without blocking the current thread.

See Also

TaskCount Property, TaskList Property, TaskAbort Method, TaskDone 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. 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.

The **Reset** and **Uninitialize** methods will abort all active background transfers and wait for those tasks to complete before returning to the caller. It is recommended that your application explicitly wait for background transfers to complete or abort them using this method before allowing the program to terminate. This will ensure that your program can perform any necessary cleanup operations. If there are active background tasks running at the time that the control instance is destroyed, it can force the control to stop those worker threads immediately without waiting for them to terminate gracefully.

See Also

Initialize Method

VerifyFile Method

Verify that the contents of a file on the local system are the same as the specified file on the server..

Syntax

object.**VerifyFile**(*LocalFile*, *RemoteFile*, [*Options*])

Parameters

LocalFile

A string that specifies the name of the file on the local system.

RemoteFile

A string that specifies the name of the file on the server.

Options

A numeric bitmask which specifies the options that may be used when comparing the files. This argument may be any one of the following values:

Value	Description
ftpVerifyDefault	File verification should use the best option available based on the available server features. If the server supports the XMD5 command, the control will calculate an MD5 hash of the local file contents and compare the value with the file on the server. If the server does not support the XMD5 command, but it does support the XCRC command, the control will calculate a CRC32 checksum of the local file contents and compare the value with the file on the server. If the server does not support either the XMD5 or XCRC commands, the control will compare the size of the local and remote files.
ftpVerifySize	Files are verified by comparing the number of bytes of data in the local and remote files. This is the least reliable method, and should only be used if the server does not support either the XMD5 or XCRC commands.
ftpVerifyCRC32	Files are verified by calculating a CRC-32 checksum of the local file contents and comparing it with the value returned by the server in response to the XCRC command. This method should only be used if the server does not support the XMD5 command.
ftpVerifyMD5	Files are verified by calculating an MD5 hash of the local file contents and comparing it with the value returned by the server in response to the XMD5 command. This is the preferred method for performing file verification.

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 **VerifyFile** method will attempt to verify that the contents of the local and remote files are identical using one of several methods, based on the features that the server supports. Preference will be given to the most reliable method available, using either an MD5 hash, a CRC-32 checksum or comparing the size of the file, in that order.

It is not recommended that you use this method with text files because of the different end-of-line conventions used by different operating systems. For example, a text file on a Windows system uses a carriage-return and linefeed pair to indicate the end of a line of text. However, on a UNIX system, a single linefeed is used to indicate the end of a line. This can cause the **VerifyFile** method to indicate the files are not identical, even though the only difference is in the end-of-line characters that are used.

See Also

BufferSize Property, Priority Property, GetData Method, GetFile Method, GetMultipleFiles Method, PutData Method, PutFile Method, OnGetFile Event, OnProgress Event

Write Method

Write data to the server.

Syntax

object.**Write**(*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 bytes actually written to the server, or -1 if an error was encountered.

Remarks

The **Write** method sends the data in ***buffer*** to the server. 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 blocking, the application will wait until the data can be sent. If the control is non-blocking and the write fails because it could not send all of the data to the server, the **OnWrite** event will be fired when the server can accept data again.

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 UTF-8 (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

IsConnected Property, IsWritable Property, Timeout Property, Read Method, OnWrite Event

File Transfer Protocol Control Events

Event	Description
OnCancel	This event is generated when a blocking operation is canceled
OnCommand	This event is generated when the server processes a command issued by the client
OnConnect	This event is generated when a connection is established
OnDisconnect	This event is generated when a connection is terminated
OnError	This event is generated when a control error occurs
OnGetFile	This event is generated when a file transfer is initiated
OnProgress	This event is generated during data transfer
OnPutFile	This event is generated when a file transfer is initiated
OnRead	This event is generated when data is available to be read
OnTaskBegin	This event is generated when a background task begins
OnTaskEnd	This event is generated when a background task completes
OnTaskRun	This event is generated while a background task is active
OnTimeout	This event is generated when a blocking operation times out
OnWrite	This event is generated when data can be written to the server

OnCancel Event

The **OnCancel** event is generated when a blocking operation is canceled.

Syntax

Sub *object*_**OnCancel** ([*Index* **As Integer**])

Remarks

This event is generated 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, OnTimeout Event

OnCommand Event

The **OnCommand** event is generated when the client sends a command to the server and receives a reply indicating the results of that command.

Syntax

Sub *object*_**OnCommand**([*Index* **As Integer**], **ByVal** *ResultCode* **As Variant**, **ByVal** *ResultString* **As Variant**)

Remarks

The **OnCommand** event is generated when the client receives a reply from the server after some action has been taken. The ***ResultCode*** argument contains the numeric result code returned by the server. The result codes returned from the server fall into one of the following categories:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

The ***ResultString*** argument contains the descriptive string returned by the server which describes the result. The string contents may vary depending on the type of server.

See Also

ResultCode Property, ResultString Property, Command Method

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 server as a result of a **Connect** method call. This event is only triggered when the **Blocking** property is set to False.

See Also

Blocking Property, Connect Method, OnDisconnect Event, OnWrite 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. This event is only triggered when the **Blocking** property is set to False.

When the **OnDisconnect** event fires, it is possible that there may still be buffered data available to read from the server. Before disconnecting from the server, the application should attempt to read any remaining data until the **Read** method returns a value of zero, or returns an error indicating that the operation would block.

See Also

Blocking Property, IsConnected Property, IsReadable Property, Connect Method, Disconnect Method, Read Method, 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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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

OnFileList Event

The **OnFileList** event is generated when a remote file list is parsed by the control. This event has been deprecated and should no longer be used in new applications.

Syntax

Sub *object*_**OnFileList**([*Index* **As Integer**], **ByVal** *FileName* **As Variant**, **ByVal** *FileLength* **As Variant**, **ByVal** *FileDate* **As Variant**, **ByVal** *FileOwner* **As Variant**, **ByVal** *FileGroup* **As Variant**, **ByVal** *FilePerms* **As Variant**, **ByVal** *IsDirectory* **As Variant**)

Remarks

The **OnFileList** event is generated as the control parses the list of files returned by the server as the result of the application calling the **FileList** method. The following arguments are passed to the event handler:

FileName

A string which specifies the name of the file that status information is being returned for.

FileLength

A numeric value which specifies the size of the file on the server. Note that if this is a text file, the file size may be different on the server than it is on the local system. This is because different operating systems use different conventions that indicate the end of a line and/or the end of the file. On MS-DOS and Windows platforms, directories have a file size of zero bytes.

FileDate

A string argument which specifies the date and time the file was created or last modified on the server. The date format that is returned is expressed in local time (in other words, the timezone of the server is not taken into account) and depends on both the local host settings via the Control Panel and the format of the date and time information returned by the server.

FileOwner

A string argument which specifies the owner of the file on the server. On some platforms, this information may not be available for security reasons if an anonymous login session was specified.

FileGroup

A string argument which specifies the group that the file owner belongs to. On some platforms, this information may not be available for security reasons if an anonymous login session was specified.

FilePerms

A numeric value which specifies the permissions assigned to the file. This value is actually a combination of one or more bit flags that specify the individual permissions for the file owner, group and world (all other users). The permissions are as follows:

Value	Description
ftpPermSymbolicLink	The file is a symbolic link to another file. Symbolic links are special types of files found on UNIX based systems which are similar to Windows shortcuts.
ftpPermOwnerRead	The owner has permission to open the file for reading. If the current user is the owner of the file, this grants the user the right to download the file to the local system.
ftpPermOwnerWrite	The owner has permission to open the file for writing. If the current user is the owner of the file, this grants the user the right

	to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermOwnerExecute	The owner has permission to execute the contents of the file. The file is typically either a binary executable, script or batch file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermGroupRead	Users in the specified group have permission to open the file for reading. If the current user is in the same group as the file owner, this grants the user the right to download the file.
ftpPermGroupWrite	Users in the specified group have permission to open the file for writing. On some platforms, this may also imply permission to delete the file. If the current user is in the same group as the file owner, this grants the user the right to replace the file. If this permission is set for a directory, this grants the user the right to create and delete files.
ftpPermGroupExecute	Users in the specified group have permission to execute the contents of the file. If this permission is set for a directory, this may also grant the user the right to open that directory and search for files in that directory.
ftpPermWorldRead	All users have permission to open the file for reading. This permission grants any user the right to download the file to the local system.
ftpPermWorldWrite	All users have permission to open the file for writing. This permission grants any user the right to replace the file. If this permission is set for a directory, this grants any user the right to create and delete files.
ftpPermWorldExecute	All users have permission to execute the contents of the file. If this permission is set for a directory, this may also grant all users the right to open that directory and search for files in that directory.

For the proprietary Sterling directory formats, the status code is returned in the ***FilePerms*** argument. This value is a combination of bits. Bits 0-25 correspond to letters of the alphabet, most of which have distinct meanings in the Sterling formats.

Letter code	Bit position	Hexadecimal value
A	1h	
B	2h	
C	4h	
n-th letter of alphabet	n-1	2 to the (n-1) power
Z	2000000h	

For the proprietary Sterling directory formats, bits 26-31 represent the transfer protocol associated with the file:

Protocol	Bit position	Hexadecimal value	Value
TCP	4000000h	ftpSterlingStatusTcp	

FTP	8000000h	ftpSterlingStatusFtp
BSC	10000000h	ftpSterlingStatusBsc
ASC	20000000h	ftpSterlingStatusAsc
FTS	40000000h	ftpSterlingStatusFts
other	80000000h	ftpSterlingStatusOther

IsDirectory

A boolean argument which specifies if the file is a directory or a regular file.

See Also

ParseList Property, FileList Method, GetFileStatus Method, OnLastFile Event

OnGetFile Event

The **OnGetFile** event is generated when a file transfer is initiated

Syntax

Sub *object*_**OnGetFile**([*Index* **As Integer**], **ByVal** *LocalFile* **As Variant**, **ByVal** *RemoteFile* **As Variant**)

Remarks

The **OnGetFile** event is generated when a file transfer is initiated by calling the **GetFile** or **GetMultipleFiles** methods. This will be followed by one or more **OnProgress** events which will indicate the progress of the transfer. If multiple files are being downloaded, this event will fire for each file as it is transferred.

See Also

GetFile Method, GetMultipleFiles Method, OnProgress Event

OnLastFile Event

The **OnLastFile** event is generated when the last file in a remote file list has been processed. This event has been deprecated and should no longer be used in new applications.

Syntax

Sub *object*_**OnLastFile** ([*Index* **As Integer**])

Remarks

The **OnLastFile** event is generated when the list file in the remote file list has been processed by the control. This event is only generated when the **ParseList** property is set to True.

See Also

ParseList Property, FileList Method, GetFileStatus Method, OnFileList Event

OnProgress Event

The **OnProgress** event is generated during data transfer.

Syntax

Sub *object*_**OnProgress** ([*Index* **As Integer**], **ByVal** *FileName* **As Variant**, **ByVal** *FileSize* **As Variant**, **ByVal** *BytesCopied* **As Variant**, **ByVal** *Percent* **As Variant**)

Remarks

The **OnProgress** event is generated during the transfer of data between the client and server, indicating the amount of data exchanged. For transfers of 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:

FileName

A string which specifies the name of the file currently being transferred. This always corresponds to the name of the file on the server.

FileSize

The size of the file being transferred in bytes. This value may be zero if the control cannot obtain the size of the file from the server. If the total number of bytes is less than 2 GiB, the value will be a **Long** (32-bit) integer. For very large transfers, it will be a **Double** floating-point value.

BytesCopied

The number of bytes that have been transferred between the client and server. If the number of bytes copied is less than 2 GiB, the value will be a **Long** (32-bit) integer. For very large transfers, it will be a **Double** floating-point value.

Percent

The percentage of data that's been transferred, expressed as an integer value between 0 and 100, inclusive. If the size of the file on the server cannot be determined, this value will always be 100.

This event is only generated when a file is transferred using the **GetFile** or **PutFile** methods, or equivalent actions. If the client is reading or writing the file data directly to the server using the **Read** or **Write** methods then the application is responsible for calculating the completion percentage and updating any user interface controls.

See Also

TransferBytes Property, TransferRate Property, TransferTime Property, GetData Method, GetFile Method, PutData Method, PutFile Method, OnGetFile Event, OnPutFile Event

OnPutFile Event

The **OnPutFile** event is generated when a file transfer is initiated.

Syntax

Sub *object*_**OnPutFile**([*Index* **As Integer**], **ByVal** *LocalFile* **As Variant**, **ByVal** *RemoteFile* **As Variant**)

Remarks

The **OnPutFile** event is generated when a file transfer is initiated by calling the **PutFile** or **PutMultipleFiles** methods. This will be followed by one or more **OnProgress** events which will indicate the progress of the transfer. If multiple files are being uploaded, this event will fire for each file as it is transferred.

See Also

PutFile Method, PutMultipleFiles Method, OnProgress Event

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 triggered when the **Blocking** property is set to False.

See Also

IsReadable Property, Read Method, Write Method, OnWrite Event

OnTaskBegin Event

The **OnTaskBegin** event occurs when a background task starts.

Syntax

Sub *object*_**OnTaskBegin** ([*Index* **As Integer**], **ByVal** *TaskId* **As Variant**)

Remarks

The **OnTaskBegin** event is generated when a background task associated with an asynchronous file transfer begins running. The arguments to this event are:

TaskId

An integer value that uniquely identifies the background task.

This event can be used in conjunction with the **OnTaskEnd** event to monitor one or more background tasks that are created to perform asynchronous file transfers. The task ID passed to this event can be used to uniquely identify the task and corresponds to the worker thread that has been created to manage the client session. The application should consider the ID to be an opaque value and never make assumptions about how an ID is assigned to a background task.

See Also

AsyncGetFile Method, AsyncPutFile Method, OnTaskEnd Event, OnTaskRun Event

OnTaskEnd Event

The **OnTaskEnd** event occurs when a background task completes.

Syntax

Sub *object*_**OnTaskEnd** ([*Index* **As Integer**], **ByVal** *TaskId* **As Variant**, **ByVal** *TimeElapsed* **As Variant**, **ByVal** *ErrorCode* **As Variant**)

Remarks

The **OnTaskEnd** event is generated when a file transfer completes and the background task has terminated. The arguments to this event are:

TaskId

An integer value that uniquely identifies the background task.

TimeElapsed

An integer value that specifies the amount of elapsed time in milliseconds.

ErrorCode

An integer value that specifies the last error code for the task.

This event can be used in conjunction with the **OnTaskBegin** event to monitor one or more background tasks that are created to perform asynchronous file transfers. The ***TimeElapsed*** parameter will specify the number of milliseconds that the background task was active. The ***ErrorCode*** parameter specifies the last error code associated with the background task. If this value is zero, that indicates that the task completed successfully. A non-zero value indicates that the task failed and the error code value identifies why the task failed.

See Also

AsyncGetFile Method, AsyncPutFile Method, OnTaskBegin Event, OnTaskRun Event

OnTaskRun Event

The **OnTaskRun** event occurs while a background task is active.

Syntax

Sub *object*_**OnTaskRun** ([*Index* **As Integer**], **ByVal** *TaskId* **As Variant**, **ByVal** *TimeElapsed* **As Variant**, **ByVal** *Completed* **As Variant**)

Remarks

The **OnTaskRun** event is generated periodically during a file transfer while the background task is active. The arguments to this event are:

TaskId

An integer value that uniquely identifies the background task.

TimeElapsed

An integer value that specifies the amount of elapsed time in milliseconds.

Completed

An integer value that specifies an estimated percentage of completion.

The rate and number of times that this event will be generated depends on the task being performed. This event is generally analogous to the **OnProgress** event for file transfers that are performed in the current working thread, however the **OnTaskRun** event will occur for each individual background task that is active. The ***TimeElapsed*** parameter specifies the amount of time that the task has been active, and the ***Completed*** parameter specifies an estimated percentage of completion. This can be used to update the user interface if needed, however it is the application's responsibility to determine which UI component (such as a **ProgressBar** control) is associated with a particular task.

See Also

AsyncGetFile Method, AsyncPutFile Method, OnTaskBegin Event, OnTaskEnd 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. This event is only triggered when the **Blocking** property is set to True.

See Also

Timeout Property, OnCancel Event

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 triggered when the **Blocking** property is set to False.

See Also

IsWritable Property, Read Method, Write Method, OnConnect Event, OnRead Event

File Transfer Server Control

Implements a server that enables the application to send and receive files using the File Transfer Protocol.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	FtpServerCtl.FtpServer
File Name	CSFTSX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.FtpServer.11
ClassID	9E92E344-1F8A-4CCD-B448-BFD9834185E1
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 959, RFC 1579, RFC 2228

Overview

This ActiveX control provides an interface for implementing an embedded, lightweight server that can be used to exchange files with a client using the standard File Transfer Protocol. The server can accept connections from any third-party application or a program developed using the SocketTools FTP ActiveX control.

The application specifies an initial server configuration by setting the relevant properties and can implement event handlers to monitor the activities of the clients that have connected to the server. The control automatically handles the standard FTP commands and requires minimal coding on the part of the application that is hosting the control. However, the application may also use event mechanism to filter specific commands or to extend the protocol by providing custom implementations of existing commands or add entirely new commands.

The server supports active and passive mode file transfers, has compatibility options for NAT router and firewall support, and provides support for secure file transfers using explicit TLS sessions. Secure connections require that a valid TLS certificate be installed on the system.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop

and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

File Transfer 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
AuthFail	Gets and sets the maximum number of authentication attempts permitted
AuthTime	Gets and sets the amount of time a client has to authenticate the session
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
ClientAccess	Gets and sets the access rights that have been granted to the client session
ClientAddress	Return the Internet address of the current client connection
ClientCount	Return the number of active client sessions connected to the server
ClientDirectory	Return the current working directory for the active client session
ClientHome	Return the home directory for the active client session
ClientHost	Return the host name that the client used to establish the connection
ClientId	Return the unique identifier for the active client session
ClientIdle	Get and set the idle timeout period for the active client session
ClientPort	Return the port number allocated by the active client connection
ClientUser	Return the user name associated with the specified client session
CommandLine	Return the complete command line issued by the client
Directory	Get and set the full path to the root directory assigned to the server
ExecTime	Get and set maximum number of seconds that the server will permit an external command to execute
ExternalAddress	Get and set the external IP address used for passive mode data transfers
HiddenFiles	Determine if the server should permit access to hidden files
Identity	Gets and sets a string that identifies the server to the client
IdleTime	Gets and sets the maximum number of seconds a client can be idle before the server terminates the session
IsActive	Determine if the server has been started
IsAnonymous	Determine if the active client session has authenticated as an anonymous user
IsAuthenticated	Determine if the active client session has been authenticated
IsInitialized	Determine if the server has been initialized
IsListening	Determine if the server is listening for client connections
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error that occurred
LocalPath	Return the full path to the local file or directory that is the target of the current command
LocalTime	Determines if the server should return file and directory times adjusted for the local timezone
LocalUser	Determines if the server should perform user authentication using the Windows local account database
LockFiles	Determines if files should be exclusively locked when a client attempts to upload or download a file
LogFile	Gets and sets the name of the server log file

LogFormat	Gets and sets the format used when updating the server log file
LogLevel	Gets and sets the level of detail included in the server log file
MaxClients	Gets and sets the maximum number of clients that are permitted to connect to the server
MaxGuests	Gets and sets the maximum number of anonymous users that are permitted to connect to the server
MaxPort	Gets and sets the maximum port number used by the server for passive data connections
MinPort	Gets and sets the minimum port number used by the server for passive data connections
MemoryUsage	Gets the amount of memory allocated for the server and all client sessions
MultiUser	Determine if the server should be started in multi-user mode
Options	Gets and sets the options used when starting the server
Priority	Gets and sets the priority assigned to the server
ReadOnly	Determine if the server should prevent clients from uploading files
Restricted	Determine if the server should be started in restricted mode, limiting client access to the server
Secure	Determine if the server should accept secure client connections
ServerAddress	Gets and sets the address that will be used by the server to listen for connections
ServerName	Gets and sets the fully qualified domain name for the server
ServerPort	Gets and sets the port number that will be used by the server to listen for connections
ServerUuid	Gets and sets the Universally Unique Identifier (UUID) associated with the server
StackSize	Gets and sets the size of the stack allocated for threads created by the server
ThrowError	Enable or disable error handling by the container of the control
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
UnixMode	Determine if the server should impersonate a UNIX-based operating system
Version	Return the current version of the object
VirtualPath	Return the virtual path to the local file or directory that is the target of the current command

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, ServerAddress Property, ServerName Property, ServerPort 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 returned IP address allocated for that connection.

Data Type

Integer (Int32)

See Also

AdapterAddress Property, ServerAddress Property, ServerName Property

AuthFail Property

Gets and sets the maximum number of authentication attempts permitted.

Syntax

object.**AuthFail** [= *attempts*]

Remarks

The **AuthFail** property value specifies the maximum number of user authentication attempts that are permitted until the server terminates the client connection. A value of zero specifies that the default configuration limit of 3 authentication attempts per login should be allowed. The maximum number of authentication attempts is 10.

Data Type

Integer (Int32)

See Also

AuthTime Property, AddUser Method, Authenticate Method, OnAuthenticate Event

AuthTime Property

Gets and sets the amount of time a client has to authenticate the session.

Syntax

object.**AuthTime** [= *seconds*]

Remarks

The **AuthTime** property value specifies the maximum number of user authentication attempts that are permitted until the server terminates the client connection. A value of zero specifies the default value of 60 seconds. If the value is non-zero, the minimum value is 20 seconds and the maximum value is 300 seconds (5 minutes). This value is used to ensure that a client has successfully authenticated itself within a limited period of time. This prevents a potential denial-of-service attack against the server where clients establish connections and hold them open without authentication. In conjunction with the **AuthFail** property, this also limits the ability of a client to attempt to probe the server for valid username and password combinations.

Data Type

Integer (Int32)

See Also

AuthFail Property, AddUser Method, Authenticate Method, OnAuthenticate Event

CertificateName Property

Gets and sets the common name for the server certificate.

Syntax

object.**CertificateName** [= *name*]

Remarks

The **CertificateName** property sets the common name or friendly name of the server certificate that should be used with secure TLS connections. The certificate must be designated as a server certificate and have a private key associated with it, otherwise the server will be unable to create the security context for the client session. This property value is only used if security has been enabled by setting the **Secure** property to **True**.

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, ServerName 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 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

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. 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

ClientAccess Property

Gets and sets the access rights that have been granted to the client session.

Syntax

object.**ClientAccess** [= *accessflags*]

Remarks

The **ClientAccess** property is used to determine all of the access permissions that are currently granted to an authenticated client session and optionally change those permissions. For a list of user access rights that can be granted to the client, see User Access Constants.

When modifying the value of this property, it is recommended that you use bitwise OR and AND operands to set and clear specific bit flags. The exception is when using the **ftpAccessDefault** permission. If you wish to reset the client session to use the default permissions based on the server configuration and client authentication, then you should assign this value directly to the **ClientAccess** property.

This property should only be accessed within an event handler such as **OnCommand** because its value is specific to the client session that raised the event. This property will always return a value of zero outside of an event handler, and an exception will be raised if you attempt to modify this property outside of an event handler.

Data Type

Integer (Int32)

Example

```
' Allow the client to execute registered programs
FtpServer1.ClientAccess = FtpServer1.ClientAccess Or ftpAccessExecute

' Prevent the client from changing its idle timeout period
FtpServer1.ClientAccess = FtpServer1.ClientAccess And Not ftpAccessIdle
```

See Also

AddUser Method, Authenticate Method, OnAuthenticate Event

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 should only be accessed within an event handler such as **OnConnect** because its value is specific to the client session that raised the event. This property will always return an empty string when accessed outside of an event handler.

Data Type

String

See Also

ClientHost Property, ClientPort Property, ServerAddress Property, 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 includes both authenticated and unauthenticated client sessions.

Data Type

Integer (Int32)

See Also

MaxClients Property, MaxGuests Property

ClientDirectory Property

Return the current working directory for the active client session.

Syntax

object.**ClientDirectory**

Remarks

The **ClientDirectory** property returns the current working directory for the active client session. Initially this value will be the absolute path on the local system that maps to an authenticated client's home directory. The client can change its current working directory using the CWD command. The **ClientHome** property will return the home directory that has been assigned to the client.

It is important to note that the current working directory for client sessions is virtual, and does not reflect the current working directory for the server process. This property should only be accessed within an event handler after the client session has been authenticated. Unauthenticated clients are not assigned a current working directory. This property will always return an empty string when accessed outside of an event handler.

Data Type

String

See Also

ClientAddress Property, ClientHome Property, ClientId Property, IsAuthenticated Property,

ClientHome Property

Return the home directory for the active client session.

Syntax

object.**ClientHome**

Remarks

The **ClientHome** property returns the home directory for the active client session. This will be the same path to the home directory specified when the **Authenticate** method was used to authenticate the client session. If a home directory was not explicitly assigned when the client was authenticated, then this property returns the default home directory that was created for the client, or the server root directory if the **MultiUser** property was set to **False** when the server was started. The **ClientDirectory** property will return the current working directory for the client.

This property should only be accessed within an event handler after the client session has been authenticated. Unauthenticated clients are not assigned a home directory. This property will always return an empty string when accessed outside of an event handler.

Data Type

String

See Also

ClientDirectory Property, IsAuthenticated Property, MultiUser Property, Authenticate Method

ClientHost Property

Return the host name that the client used to establish the connection.

Syntax

object.**ClientHost**

Remarks

The **ClientHost** property returns the host name that the client used to establish the connection. If the client sends the HOST command, this property will return the value specified by the client. If the client does not explicitly specify the host name, then this property will return the same host name that was assigned to the server when it started.

Data Type

String

See Also

ClientAddress Property, ClientPort Property, ServerName Property, OnConnect Event

ClientId Property

Return the unique identifier for the active client session.

Syntax

object.**ClientId**

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value. This value is by the application to identify that client session, and is different than the socket handle allocated for the client. Client IDs are unique throughout the life of the server session and are never duplicated.

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. This property will always return a value of zero when accessed outside of an event handler.

Data Type

Integer (Int32)

See Also

ClientAddress Property, ClientHost Property, ServerAddress Property, ServerPort Property

ClientIdle Property

Gets and sets the maximum number of seconds a client can be idle before the server terminates the session.

Syntax

object.**ClientIdle** [= *seconds*]

Remarks

The **ClientIdle** property returns the maximum number of seconds that the active client session may be idle before the server closes the control connection. The idle timeout period for each client session is based on the value of the **IdleTime** property when the server was started, with the default value of 900 seconds (15 minutes). Changing this value inside an event handler will change the timeout period for the active client session. Clients may also use the SITE IDLE command to request that the server change the idle timeout period.

This property should only be accessed within an event handler such as **OnConnect** or **OnLogin** because its value is specific to the client session that raised the event. This property will always return a value of zero outside of an event handler, and an exception will be raised if you attempt to modify this property outside of an event handler.

When the timeout period for the client has elapsed, the **OnTimeout** event will fire prior to the client being disconnected from the server.

Data Type

Integer (Int32)

See Also

IdleTime Property, OnTimeout Event

ClientPort Property

Return the port number allocated by the active 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

ClientUser Property

Return the user name associated with the specified client session.

Syntax

object.**ClientUser**

Remarks

The **ClientUser** property returns the user name that the client used to authenticate the client session. This property should only be accessed within an event handler after the client session has been authenticated. Unauthenticated clients are not assigned a user name. This property will always return an empty string when accessed outside of an event handler.

Data Type

String

See Also

ClientAddress Property, ClientHome Property, Authenticate Method, OnAuthenticate Event

CommandLine Property

Return the complete command line issued by the client.

Syntax

object.**CommandLine**

Remarks

The **CommandLine** property is used to obtain the command that was issued by the client, and is commonly used inside **OnCommand** and **OnResult** event handlers to pre-process and post-process client commands, respectively. If the command sent by the client is used to perform an action on a file or directory, use the **LocalPath** property to get the full path to the local file that is the target of the command.

This property should only be accessed within an event handler because its value is specific to the client session that raised the event. This property will always return an empty string when accessed outside of an event handler.

Data Type

String

See Also

LocalPath Property, VirtualPath Property, OnCommand Event, OnResult Event

Directory Property

Get and set the full path to the root directory assigned to the server.

Syntax

object.**Directory** [= *pathname*]

Remarks

The **Directory** property returns the path to the root directory for the server. If this property is set to the name of a valid directory before the server is started, that directory will be considered the root directory for the server. If this property is not set, or is set to an empty string, then the server will use the current working directory as its root directory, however this is not recommended. It is recommended that you specify an absolute path to the directory, otherwise the path will be relative to the current working directory. You may include environment variables in the path surrounded by percent (%) symbols and they will be expanded.

If you have configured the server to permit clients to upload files, you must ensure that your application has permission to create files in the directory that you specify. A recommended location for the server root directory would be a subdirectory of the %ALLUSERSPROFILE% directory. Using the environment variable ensures that your server will work correctly on different versions of Windows. If the root directory does not exist at the time that the server is started, it will be created.

If the **MultiUser** property is **False**, all authenticated clients will have their current working directory initialized to the server root directory. If the **MultiUser** property is **True**, then the Public and User subdirectories will be created in the root directory, and each authenticated client will have their current working directory initialized to their individual home directory.

This property can be read after the server has started and it will return the full path to the root directory. However, attempting to change the value of this property after the server has started will cause an exception to be raised. To change the root directory for the server, you must first call the **Stop** method which will terminate all active client connections.

Data Type

String

Example

```
' Set the server root directory
FtpServer1.Directory = "%ALLUSERSPROFILE%\MyProgram\FileServer"
```

See Also

ClientDirectory Property, ClientHome Property, MultiUser Property

ExecTime Property

Get and set maximum number of seconds that the server will permit an external command to execute.

Syntax

object.**ExecTime** [= *seconds*]

Remarks

The **ExecTime** property specifies the maximum number of seconds that an external program is permitted to run on the server. External programs are registered using the **RegisterProgram** method, and are executed by the client sending the SITE EXEC command to the server. If this value is zero, the default timeout period of 5 seconds will be used. The minimum execution time is 1 second and the maximum time limit is 30 seconds.

Data Type

Integer (Int32)

See Also

AuthTime Property, IdleTime Property, RegisterProgram Method, OnExecute Event

ExternalAddress Property

Get and set the external IP address used for passive mode data transfers.

Syntax

object.**ExternalAddress**[= *ipaddress*]

Remarks

When using passive mode file transfers, the server creates a second listening (passive) socket that is used to exchange data between the client and server. The client sends the PASV command, and the server responds with its IP address and the ephemeral port number that was selected for the transfer. If the server is located behind a router that performs Network Address Translation (NAT), the address that the server will return will typically be a non-routable IP address assigned to the local system on the LAN side of the network. Setting the **ExternalAddress** property will instruct the server to return a different IP address in response to the PASV command sent by the client. Typically you would use the address assigned to the router on the Internet side of the connection.

If the **ExternalAddresss** property is not assigned a specific address, reading this property value will cause the control to automatically determine its external IP address. This 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.

This property will not change the IP address the server is using to listen for client connections. The only way to change the listening IP address is to stop and restart the server using the new address. This property only changes the IP address that is reported to clients when a passive data connection is used. Incorrect use of this property can prevent the client from establishing a data connection to the server. The address must be in the same address family as the local address that the server was started with. For example, if the server was started using an IPv4 address, the IP address assigned to this property cannot be an IPv6 address.

Data Type

String

See Also

ClientAddress Property, ServerAddress Property

HiddenFiles Property

Determine if the server should permit access to hidden files.

Syntax

object.**HiddenFiles** [= { True | False }]

Remarks

The **HiddenFiles** property determines if the server should allow clients to access files with the hidden and/or system attribute. If this property is **True**, then hidden files are included in directory listings and clients may download or replace hidden files. If the property is **False**, hidden files are not included in directory listings and any attempt to access, delete or modify a hidden file will result in an error.

The default value for this property is **False**.

Data Type

Boolean

See Also

ReadOnly Property, Restricted Property, Start Method

Identity Property

Gets and sets a string that identifies the server to the client.

Syntax

object.**Identity** [= *description*]

Remarks

The **Identity** property returns a string that is used to identify the server. It is used for informational purposes only and does not affect the operation of the server. Typically the string specifies the name of the application and a version number, and is displayed whenever a client establishes its initial connection to the server. This property can be set to assign an identity to the server, however after the server has started this property becomes read-only.

Data Type

String

See Also

ClientAddress Property, ClientPort Property, ServerName Property, OnConnect Event

IdleTime Property

Gets and sets the maximum number of seconds a client can be idle before the server terminates the session.

Syntax

object.**IdleTime** [= *seconds*]

Remarks

The **IdleTime** property specifies the maximum number of seconds that a client session may be idle before the server closes the control connection to the client. A value of zero specifies the default value of 900 seconds (15 minutes). If the value is non-zero, the minimum value is 60 seconds and the maximum value is 7200 seconds (2 hours). This value is used to initialize the default idle timeout period for each client session. A client may request that the server change the idle timeout period for its session by sending the SITE IDLE command. The server determines if a client is idle based on the time the last command was issued and whether or not a file transfer is in progress.

The **ClientIdle** property can be used to determine the idle timeout period for a specific client. When the timeout period for the client has elapsed, the **OnTimeout** event will fire prior to the client being disconnected from the server.

Data Type

Integer (Int32)

See Also

ClientIdle Property, OnTimeout Event

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

IsAnonymous Property

Determine if the active client session has authenticated as an anonymous user.

Syntax

object.**IsAnonymous**

Remarks

The **IsAnonymous** property returns **True** if the active client session has authenticated as an anonymous (guest) user. This property should only be accessed within an event handler such as **OnCommand** because its value is specific to the client session that raised the event. This property will always return a value of **False** outside of an event handler.

Data Type

Boolean

See Also

IsAuthenticated Property, IsListening Property, Authenticate Method

IsAuthenticated Property

Determine if the active client session has been authenticated.

Syntax

object.**IsAuthenticated**

Remarks

The **IsAuthenticated** property returns **True** if the active client session has successfully authenticated with a valid username and password. This property should only be accessed within an event handler such as **OnCommand** because its value is specific to the client session that raised the event. This property will always return a value of **False** outside of an event handler.

Data Type

Boolean

See Also

IsAnonymous Property, IsListening Property, Authenticate Method, OnAuthenticate Event

IsInitialized Property

Determine if the server has been initialized.

Syntax

object.**IsInitialized**

Remarks

The **IsInitialized** property is used to determine if the current instance of the server 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, Start Method, Stop 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.

Data Type

Boolean

See Also

IsActive Property, Start Method, Stop Method

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

LocalPath Property

Return the full path to the local file or directory that is the target of the current command.

Syntax

object.**LocalPath** [= *filename*]

Remarks

The **LocalPath** property returns the full path to a local file name or directory specified by the client as an argument to a standard FTP command. For example, if the client sends the RETR command to the server, this property will return the complete path to the local file that the client wants to download. This property will only return a value for those standard commands that perform some action on a file or directory, otherwise it will return an empty string.

Setting this property allows you to effectively redirect the client to use a different file than the one that was actually requested. If the path is absolute, then it will be used as-is. If the path is relative, it will be relative to the current working directory for the active client session. The full path to this file is not limited to the server root directory or its subdirectory, it can specify a file anywhere on the local system. If this property is set to an empty string, then the server will revert to using the actual file or directory name specified by the command.

This property should only be set within an **OnCommand** event handler, and only for those commands that perform an action on a file or directory. If the current command does not target a file or directory, setting this property will cause an exception to be raised by the control. Exercise caution when using this property to redirect the server to use a different file than the one requested by the client; changing the target file may cause the client to behave in unexpected ways.

Data Type

String

See Also

VirtualPath Property, ResolvePath Method, OnCommand Event

LocalTime Property

Determines if the server should return file and directory times adjusted for the local timezone.

Syntax

object.**LocalTime** [= { True | False }]

Remarks

The **LocalTime** property determines if the server should return file and directory times adjusted for the local timezone. By default, the server will return all file times as UTC values. This option affects the time information sent to a client when a list of files is requested, as well as when status information is requested for a specific file. This property value will not affect the MDTM and MFMT commands which always use file times as UTC values.

The default value for this property is **False**.

Data Type

Boolean

See Also

HiddenFiles Property, LockFiles Property, ReadOnly Property

LocalUser Property

Determines if the server should perform user authentication using the Windows local account database.

Syntax

object.**LocalUser** [= { True | False }]

Remarks

The **LocalUser** property determines if the server should perform user authentication using the Windows local account database. If this option is not specified, the application is responsible for creating virtual users using the **AddUser** method or implementing an **OnAuthenticate** event handler and authenticating client sessions individually.

If this property is set to **True**, a client can authenticate as a local user, however the session will not inherit that user's access rights. All files that are accessed or created by the server will continue to use the permissions of the process that started the server. For example, consider a server application that was started by local user ***A***. Next, a client connects to the server and authenticates itself as local user ***B***. When that client uploads a file to the server, the file that is created will be owned by user ***A***, not user ***B***. This ensures that the server application retains ownership and control of the files that have been created or modified.

The default value for this property is **False**.

Data Type

Boolean

See Also

IsAuthenticated Property, AddUser Method, Authenticate Method, OnAuthenticate Event

LockFiles Property

Determines if files should be exclusively locked when a client attempts to upload or download a file.

Syntax

object.**LockFiles** [= { True | False }]

Remarks

The **LocalTime** property determines if files should be exclusively locked when a client attempts to upload or download a file. If another client attempts to access the same file, the operation will fail. By default, the server will permit multiple clients to access the same file, although it will still write-lock files that are in the process of being uploaded..

The default value for this property is **False**.

Data Type

Boolean

See Also

LocalTime Property, HiddenFiles Property, ReadOnly Property

LogFile Property

Gets and sets the name of the server log file.

Syntax

object.**LogFile** [= *filename*]

Remarks

The **LogFile** property is used to specify the name of a file that will contain a log of all client activity. The **LogFormat** and **LogLevel** properties affect the specific format for the file and the level of detail included in the log. It is recommended that you specify an absolute path to the log file, otherwise the path will be relative to the current working directory. You may include environment variables in the path surrounded by percent (%) symbols and they will be expanded.

If the log file does not exist it will be created when the server is started. If file already exists, the server will append the new logging data to the file. The server must have permission to create and/or modify the specified file.

Setting this property to an empty string after the server has been started will have the effect of disabling logging, setting the logging level to 0 and the logging format to **ftpLogNone**.

Data Type

String

Example

```
' Enable server logging
FtpServer1.LogFile = "%ALLUSERSPROFILE%\MyProgram\Server.log"
FtpServer1.LogFormat = ftpLogExtended
FtpServer1.LogLevel = 5
```

See Also

LogFormat Property, LogLevel Property

LogFormat Property

Gets and sets the format used when updating the server log file.

Syntax

object.**LogFormat** [= *format*]

Remarks

The **LogFormat** property is used to specify the format of the server log file. It may be one of the following values:

Value	Description
ftpLogNone	This value specifies that the server should not create or update a log file. This is the default property value.
ftpLogCommon	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about file transfers.
ftpLogExtended	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session.

Data Type

Integer (Int32)

Example

```
' Enable server logging
FtpServer1.LogFile = "%ALLUSERSPROFILE%\MyProgram\Server.log"
FtpServer1.LogFormat = ftpLogExtended
FtpServer1.LogLevel = 5
```

See Also

LogFile Property, LogLevel Property

LogLevel Property

Gets and sets the level of detail included in the server log file.

Syntax

object.**LogLevel** [= *level*]

Remarks

The **LogLevel** property is used to specify the level of detail that should generated in the log file. The minimum value is 1 and the maximum value is 10. If this parameter is zero, it is the same as specifying a log file format of **ftpLogNone** and will disable logging by the server

Data Type

Integer (Int32)

Example

```
' Enable server logging
FtpServer1.LogFile = "%ALLUSERSPROFILE%\MyProgram\Server.log"
FtpServer1.LogFormat = ftpLogExtended
FtpServer1.LogLevel = 5
```

See Also

LogFile Property, LogFormat Property

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.

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. You can also use the **Throttle** method to change the maximum number of guest users, the maximum number of clients per IP address and the rate at which clients can connect to 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.

The default value for this property is 100 client connections.

Data Type

Integer (Int32)

See Also

MaxGuests Property, Start Method, Throttle Method

MaxGuests Property

Gets and sets the maximum number of anonymous users that are permitted to connect to the server.

Syntax

object.**MaxGuests** [= *guests*]

Remarks

The **MaxGuests** property specifies the maximum number of guest users 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 guest users drops below the specified value. A guest user is one that authenticates with the username "anonymous" and their email address as the password.

Changing the value of this property while a server is actively listening for connections will modify the maximum number of guest logins permitted, but it will not affect connections that have already been established. You can also use the **Throttle** method to change the maximum number of clients, the maximum number of clients per IP address and the rate at which clients can connect to the server.

The default value for this property is zero, disabling guest logins. If your server is accessible to the public and you decide to allow guest users, it is recommended that you set the **Restricted** property to True, and you should not grant permission for guests to upload files or execute registered programs using the SITE EXEC command.

Data Type

Integer (Int32)

See Also

MaxClients Property, Restricted Property, Start Method, Throttle Method

MaxPort Property

Gets and sets the maximum port number used by the server for passive data connections.

Syntax

object.**MaxPort** [= *port*]

Remarks

The **MaxPort** property specifies the maximum range of port numbers that will be used with passive data connections. A value of zero specifies the default value of 65535 should be used. The minimum value of this member is 10000 and the maximum value is 65535. If the value is non-zero, it must be greater than the value of the **MinPort** property.

Attempting to change the value of this property after the server has started will cause an exception to be raised. To change the maximum port number for the server, you must first call the **Stop** method which will terminate all active client connections.

Data Type

Integer (Int32)

See Also

ExternalAddress Property, MinPort Property

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

MinPort Property

Gets and sets the maximum port number used by the server for passive data connections.

Syntax

object.**MinPort** [= *port*]

Remarks

The **MinPort** property specifies the minimum range of port numbers that will be used with passive data connections. A value of zero specifies that the default value of 30000 should be used. The minimum value of this member is 10000 and the maximum value is 65535. If the value is non-zero, it must be less than the value of the **MaxPort** property.

Attempting to change the value of this property after the server has started will cause an exception to be raised. To change the minimum port number for the server, you must first call the **Stop** method which will terminate all active client connections.

Data Type

Integer (Int32)

See Also

ExternalAddress Property, MaxPort Property

MultiUser Property

Determine if the server should be started in multi-user mode.

Syntax

object.**MultiUser** [= { True | False }]

Remarks

The **MultiUser** property determines if the server should be started in multi-user mode. If this property is set to **True**, each user will be assigned their own home directory which will be based on their user name. When a client authenticates as that user, its current working directory is set to the user's home directory. If this property is set to **False**, then all users will share the server root directory by default. This property does not affect the maximum number of simultaneous client connections to the server. To isolate users to their own individual home directory, set the **Restricted** property to **True**.

Setting this property to **True** will cause the server to create two subdirectories under the server root directory named Public and Users. The Public subdirectory is where public files should be stored, and also serves as the home directory for anonymous (guest) users. The Users subdirectory is where the home directories for each user will be created.

Attempting to change the value of this property after the server has started will cause an exception to be raised. To change this property value, you must first call the **Stop** method which will terminate all active client connections.

The default value for this property is **False**.

Data Type

Boolean

See Also

Directory Property, ReadOnly Property, Restricted Property, Start Method

Options Property

Gets and sets the options used when creating an instance of the server.

Syntax

object.**Options** [= *value*]

Remarks

The **Options** property is an integer value which specifies one or more options. The value specified for this property will be used as the default options when starting the server. The property value is created by using a bitwise operator with one or more of the following values:

Value	Description
ftpServerMultiUser	This option specifies the server should be started in multi-user mode, where users are provided with their own home directories based on their username. If this option is not specified, then all users will share the server root directory by default. This option does not affect the maximum number of simultaneous client connections to the server. To isolate users to their own individual home directory, combine this option with the **ftpServerRestricted** option.
ftpServerRestricted	This option specifies the server should be initialized in a restricted mode that isolates the server and limits the ability for clients to access files on the host system. All file transfers are limited to the user's home directory. This option also disables certain site-specific commands. This is a recommended option for general purpose applications designed to accept connections from clients over the Internet. This option is only meaningful if the **ftpServerMultiUser** option has also been specified. All clients are restricted to the server root directory and its subdirectories, regardless of whether this option is specified or not.
ftpServerLocalUser	This option specifies the server should perform user authentication using the Windows local account database. This option is useful if the server should accept local usernames, or if the application does not wish to implement an event handler for user authentication. If this option is not specified, the application is responsible for authenticating all users.
ftpServerAnonymous	This option specifies the server should accept anonymous client connections. This is typically used to provide public access to files without requiring the client to have valid credentials on the server. Anonymous clients are automatically authenticated by the server, but are restricted to a public directory and subdirectories. If this option is enabled, it is recommended that you also specify the **ftpServerReadOnly** option to prevent anonymous clients from uploading files to the server.
ftpServerReadOnly	This option specifies the server should only allow read-only access to files by default. If this option is enabled, it will change the default permissions granted to authenticated users.

	Anonymous clients will not be able to upload, rename or delete files and cannot create subdirectories. It is recommended that this option be enabled if the server is publicly accessible over the Internet and the **ftpServerAnonymous** option has been specified.
ftpServerLocalTime	This option specifies the server should return file and directory times adjusted for the local timezone. By default, the server will return all file times as UTC values. This option affects the time information sent to a client when a list of files is requested, as well as when status information is requested for a specific file. This option will not affect the MDTM and MFMT commands which always use file times as UTC values.
ftpServerLockFiles	This option specifies that files should be exclusively locked when a client attempts to upload or download a file. If another client attempts to access the same file, the operation will fail. By default, the server will permit multiple clients to access the same file, although it will still write-lock files that are in the process of being uploaded.
ftpServerHiddenFiles	This option specifies that when a client requests a list of files in a directory, the server should include any hidden and system files files or subdirectories. By default, the server will not include hidden or system files, although they are still accessible to the client if it knows the name of the file. File names that begin with a period are also considered to be hidden files and will not normally be included in file listings.
ftpServerUnixMode	This option specifies the server should impersonate a UNIX-based operating system. The server will identify itself as running on a UNIX system and directory listings will be in a format commonly used by UNIX. If this option is not specified, the server will identify itself as running on Windows NT and directory listings will be in the same format used by the Microsoft IIS FTP server. Note that this option does not affect the path delimiter used with file and directory names.
ftpServerExternal	This option specifies the server is listening for client connections from behind a router that uses Network Address Translation (NAT). If enabled, the server will report its external IP address rather than the address assigned to it on the local network. For the server to accept connections from behind a NAT router, the router must be configured to direct inbound traffic to the specified port number on the host system.
ftpServerSecure	This option specifies that secure connections using TLS should be enabled. If neither the **ftpServerExplicitTLS** or **ftpServerImplicitTLS** options are specified, the server automatically determines the appropriate type based on the port number. If the local port number is 990, then implicit TLS will be used, otherwise explicit TLS will be used. This option requires that a valid TLS certificate be installed on the local host.

ftpServerExplicitTLS	This option specifies the server will accept the AUTH TLS command and negotiate a secure connection with the client after that command is issued. This option implies the **ftpServerSecure** option and requires that a valid TLS certificate be installed on the local host.
ftpServerImplicitTLS	This option specifies the server should negotiate a secure connection with the client immediately after it connects to the server. It is recommended that you only use this option if the server is listening for connections on port 990, which is the standard port for FTP servers using implicit TLS. This option implies the **ftpServerSecure** option and requires that a valid TLS certificate be installed on the local host.
ftpServerSecureFallback	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.

Most of these options have a corresponding Boolean property. For example, the **ftpServerRestricted** option corresponds to the **Restricted** property, where setting the property to True enables the option and setting it to False disables the option.

In most cases, it is recommended that you use the property value related to the option, rather than setting the **Options** property. It will make your code more readable and prevent potential compatibility issues with subsequent versions of the control. If you do decide to specify option bit flags, it is recommended that you use the constant name rather than the numeric value.

Data Type

Integer (Int32)

See Also

HiddenFiles Property, LocalTime Property, LocalUser Property, LockFiles Property, MaxGuests Property, MultiUser Property, ReadOnly Property, Restricted Property, Secure Property, UnixMode Property, Start Method

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
ftpPriorityBackground	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.
ftpPriorityLow	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.
ftpPriorityNormal	The default priority which balances resource and processor utilization. It is recommended that most applications use this priority.
ftpPriorityHigh	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.
ftpPriorityCritical	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 **ftpPriorityNormal** 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

Start Method

ReadOnly Property

Determine if the server should prevent clients from uploading files.

Syntax

object.**ReadOnly** [= { True | False }]

Remarks

The **ReadOnly** property determines if the server should only allow read-only access to files by default, changing the default permissions granted to authenticated users. If this property is set to **True**, anonymous users will not be able to upload, rename or delete files and cannot create subdirectories. This is recommended if the server is publicly accessible over the Internet and guest logins are permitted.

Attempting to change the value of this property after the server has started will cause an exception to be raised. To change this property value, you must first call the **Stop** method which will terminate all active client connections.

The default value for this property is **False**.

Data Type

Boolean

See Also

Directory Property, ReadOnly Property, Restricted Property, Start Method

Restricted Property

Determine if the server should be started in restricted mode, limiting client access to the server.

Syntax

object.**Restricted** [= { True | False }]

Remarks

The **Restricted** property determines if the server should be initialized in a restricted mode that isolates the server and limits the ability for clients to access files on the host system. If this property is set to **True**, all file transfers are limited to the user's home directory and certain site-specific commands are disabled. This is recommended for general purpose applications designed to accept connections from clients over the Internet. This property value is only meaningful if the **MultiUser** property has also been set to **True**.

Attempting to change the value of this property after the server has started will cause an exception to be raised. To change this property value, you must first call the **Stop** method which will terminate all active client connections.

The default value for this property is **False**.

Data Type

Boolean

See Also

Directory Property, MultiUser Property, Restricted Property, Start 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.

Data Type

Boolean

See Also

CertificateName Property, CertificateStore 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. By default the server will accept connections on any appropriately configured network adapter. 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.

If an IPv6 address is specified as the server 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.

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

Gets and sets the fully qualified domain name for the server.

Syntax

object.**ServerName** [= *hostname*]

Remarks

The **ServerName** property returns the fully qualified domain name assigned to the server. 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.

Setting this property assigns the default hostname for the server which is reported to the client when it first establishes the connection. If the server is publicly accessible over the Internet, this property should be set to the same hostname that is associated with the server IP address.

Attempting to change the value of this property after the server has started will cause an exception to be raised. To change this property value, you must first call the **Stop** method which will terminate all active client connections.

Data Type

String

See Also

ExternalAddress Property, ServerAddress Property, ServerPort Property, Start 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. The default port number for standard connections is 21.

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. Attempting to change the value of this property after the server has started will cause an exception to be raised. To change this property value, you must first call the **Stop** method which will terminate all active client connections.

Data Type

Integer (Int32)

See Also

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

ServerUuid Property

Gets and sets the Universally Unique Identifier (UUID) associated with the server.

Syntax

object.**ServerUuid** [= *uuid*]

Remarks

The **ServerUuid** property returns the UUID that uniquely identifies this instance of the server. If the application does not set this property, a temporary UUID will be assigned to the server. If a value is assigned to this property, it must be a valid UUID string. A permanent UUID can be generated using a utility such as **uuidgen** which is included with Visual Studio.

Attempting to change the value of this property after the server has started will cause an exception to be raised. To change this property value, you must first call the **Stop** method which will terminate all active client connections.

Data Type

String

See Also

ServerAddress Property, ServerName Property, ServerPort Property, Start Method

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

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to **True**, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

See Also

LastError Property, OnError Event

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the tracing of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP client and server controls, and you set the **Trace** property to **True** on the FTP client control, function calls made by both controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named CSTRACE.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
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).

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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing 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
stTraceInfo	All function calls are written to the trace file. This is the default value.
stTraceError	Only those function calls which fail are recorded in the trace file.
stTraceWarning	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file.
stTraceHexDump	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.

Since socket function tracing is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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 the error WSAEWOULDBLOCK is returned, a warning is generated since the application simply needs to attempt to write the data at a later time.

Data Type

String

See Also

Trace Property, TraceFile Property

UnixMode Property

Determine if the server should impersonate a UNIX-based operating system.

Syntax

object.**UnixMode** [= { True | False }]

Remarks

The **UnixMode** property determines if the server should impersonate a UNIX-based operating system. If this property is set to a value of **True**, the server will identify itself as running on a UNIX system and directory listings will be in a format commonly used by UNIX. If this property value is **False**, the server will identify itself as running on Windows NT and directory listings will be in the same format used by the Microsoft IIS FTP server. Note that this option does not affect the path delimiter used with file and directory names.

Attempting to change the value of this property after the server has started will cause an exception to be raised. To change this property value, you must first call the **Stop** method which will terminate all active client connections.

The default value for this property is **False**.

Data Type

Boolean

See Also

LocalTime Property, MultiUser Property, Start Method

Version Property

Return the current version of the object.

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

VirtualPath Property

Return the virtual path to the local file or directory that is the target of the current command.

Syntax

object.**VirtualPath** [= *filename*]

Remarks

The **VirtualPath** property returns the virtual path to a local file name or directory specified by the client as an argument to a standard FTP command. For example, if the client sends the RETR command to the server, this property will return the complete virtual path to the file that the client wants to download. This property will only return a value for those standard commands that perform some action on a file or directory, otherwise it will return an empty string.

Setting this property allows you to effectively redirect the client to use a different file than the one that was actually requested. If the path is absolute, then it will be used as-is. If the path is relative, it will be relative to the current working directory for the active client session. If this property is set to an empty string, then the server will revert to using the actual file or directory name specified by the command.

This property should only be set within an **OnCommand** event handler, and only for those commands that perform an action on a file or directory. If the current command does not target a file or directory, setting this property will cause an exception to be raised by the control. Exercise caution when using this property to redirect the server to use a different file than the one requested by the client; changing the target file may cause the client to behave in unexpected ways.

Data Type

String

See Also

LocalPath Property, ResolvePath Method, OnCommand Event

File Transfer Server Control Methods

Method	Description
AddUser	Add a new virtual user to the server
Authenticate	Authenticate the client and assign access rights for the session
DeleteUser	Remove a virtual user from the server
Disconnect	Disconnect the specified client session from the server
Initialize	Initialize the control and validate the runtime license key
RegisterProgram	Register a program for use with the SITE EXEC command
Reset	Reset the internal state of the control to its default values
ResolvePath	Resolve a path to its full virtual or local file name
Restart	Restart the server, terminating all active client connections
Resume	Resume accepting new client connections
SendResponse	Send a result code and message to the client in response to a command
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

AddUser Method

Add a new virtual user to the server.

Syntax

object.**AddUser**(*UserName*, *Password*, [*AccessFlags*], [*Directory*])

Parameters

UserName

A string which specifies the user name. The maximum length of a username is 63 characters and it is recommended that names be limited to alphanumeric characters. Whitespace, control characters and certain symbols such as path delimiters and wildcard characters are not permitted. If an invalid character is included in the name, the method will fail with an error indicating the username is invalid. The username must be at least three characters in length. Usernames are not case sensitive.

Password

A string which specifies the user password. The maximum length of a password is 63 characters and is limited to printable characters. Whitespace and control characters are not permitted. If an invalid character is included in the password, the method will fail with an error indicating the password is invalid. The password must be at least one character in length. Passwords are case sensitive.

AccessFlags

An optional integer value which specifies the access clients will be given when authenticated as this user. This value created from one or more bit flags. For a list of user access permissions, see User Access Constants. If this parameter is omitted, the user is assigned default access permissions based on the server configuration.

Directory

An optional string which specifies the directory that will be the virtual user's home directory. If the server was started in multi-user mode, this directory will be relative to the user directory created by the server, otherwise it will be relative to the server root directory. If the directory does not exist, it will be created the first time that the virtual user successfully logs in to the server. If this parameter is omitted or is an empty string, a default home directory will be created for the virtual user.

Return Value

A value of zero is returned if the virtual user was created. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **AddUser** method creates a virtual user that is associated with the server. When a client connects with the server and provides authentication credentials, the server will check if the username has been created using this method. If a match is found, the client access rights will be updated.

If you wish to modify the information for an existing user, it is not necessary to delete the username first. If this method is called with a username that already exists, that record is replaced with the values passed to this method. You cannot use this method to create a virtual user named "anonymous".

The virtual users created by this method exist only as long as the server is active. If you wish to maintain a persistent database of users and passwords, you are responsible for its implementation based on the requirements of your specific application. For example, a simple implementation would be to store the user information in a local XML or INI file and then read that configuration file after the server has started, calling this method for each user that is listed.

See Also

Authenticate Method, DeleteUser Method, OnAuthenticate Event

Authenticate Method

Authenticate the client and assign access rights for the session.

Syntax

object.**Authenticate**(*ClientId*, [*AccessFlags*], [*Directory*])

Parameters

ClientId

An integer that identifies the client session.

AccessFlags

An optional integer value which specifies the access clients will be given when authenticated as this user. This value created from one or more bit flags. For a list of user access permissions, see User Access Constants. If this parameter is omitted, the client is authenticated using the default access permissions based on the server configuration.

Directory

An optional string which specifies the directory that will be the client's home directory. If the server was started in multi-user mode, this directory will be relative to the user directory created by the server, otherwise it will be relative to the server root directory. If the directory does not exist, it will be created. If this parameter is omitted or is an empty string, a default home directory will be created for the client.

Return Value

A value of zero is returned if the client session was authenticated. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Authenticate** method authenticates a client session, typically in response to an **OnAuthenticate** event that indicates a client has requested authentication. It is recommended that most applications specify **ftpAccessDefault** as the *AccessFlags* parameter for a client session, since this allows the server automatically grant the appropriate access based on the server configuration options for normal and anonymous users. If the server is going to be publicly accessible or third-party FTP clients will be used to access the server, you should always grant the **ftpAccessList** permission to clients. Many client applications will not function correctly if they are unable to obtain a list of files in the user's home directory.

If the server was started with the **MultiUser** and **Restricted** properties set to a value of **True**, the client session will be effectively locked to its home directory and cannot navigate to the server root directory. By default, restricted client sessions are also limited to only downloading files and requesting directory listings. If a client session is not restricted, the client can access files outside of its home directory. Regardless of this option, a client cannot access files outside of the server root directory.

If the **Restricted** property is **True** or the **ftpAccessAnonymous** permission is specified, the client session will be authenticated in a restricted mode and the access rights for the session will persist until the client disconnects from the server. Unlike regular users, the access rights for a restricted client cannot be changed by the server at a later point. This restriction is designed to prevent the inadvertent granting of rights to an untrusted client that could compromise the security of the server.

If the *Directory* parameter is omitted or is an empty string and the server has been started in multi-user mode, each user is assigned their own home directory based on their username. If the server has not been started in multi-user mode, then the default home directory will be the server root directory

and is shared by all users. The **ClientHome** property will return the full path to the home directory for an authenticated client.

If the **ftpAccessExecute** permission is granted to the client session, it can execute external programs using the SITE EXEC command. Because the program is executed in the context of the server process, it is recommended that you limit access to this functionality and ensure that the programs being executed do not introduce any security risks to the operating system. This permission is never granted by default, and the SITE EXEC command will return an error if the client session is anonymous, regardless of whether this permission is granted or not.

This method is should only be used for custom authentication schemes and is not necessary if you have used the **AddUser** method to create virtual users.

See Also

MultiUser Property, Restricted Property, AddUser Method, DeleteUser Method, OnAuthenticate Event

DeleteUser Method

Remove a virtual user from the server.

Syntax

object.**DeleteUser**(*UserName*)

Parameters

UserName

A string which specifies the user name to be deleted. Usernames are not case sensitive.

Return Value

A value of zero is returned if the virtual user was deleted. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **DeleteUser** method removes a virtual user that was created by a previous call to the **AddUser** method. This method will not match partial usernames and wildcard characters cannot be used to delete multiple users. Usernames are not case sensitive. You cannot use this method to delete the "anonymous" user.

See Also

AddUser Method, Authenticate Method, OnAuthenticate Event

Disconnect Method

Disconnect the specified client session from the server.

Syntax

object.**Disconnect**(*ClientId*)

Parameters

ClientId

An integer that identifies the client session.

Return Value

A value of zero is returned if the client was signaled to terminate its connection to the server. 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 other resources that were 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 disconnect the client.

The thread that is managing the client will be signaled that it should disconnect from the server, and it will begin the process of terminating the session. This is an asynchronous process and it is not guaranteed that the client will have actually disconnected from the server at the time that this method returns to the caller.

See Also

Start Method, Stop Method, OnConnect Event, OnDisconnect Event

Initialize Method

Initialize the server 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 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

```
Dim objServer As Object
Set objServer = CreateObject("SocketTools.FtpServer.11")

nError = objServer.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize the SocketTools object"
    End
End If
```

See Also

IsInitialized Property, Start Method, Stop Method, Uninitialize Method

RegisterProgram Method

Register a program for use with the SITE EXEC command.

Syntax

object.**RegisterProgram**(*CommandName*, *ProgramFile*, [*Parameters*], [*Directory*])

Parameters

CommandName

A string which identifies the external program. This is the name that is passed to the SITE EXEC command and does not need to match the actual name of the executable file on the local system. The maximum length of the command name is 31 characters and this parameter cannot be an empty string.

ProgramFiles

A string which specifies the full path to the executable program on the local system.

Parameters

An optional string that specifies additional parameters for the program. This value will be passed to the program as command line arguments. If the program does not require any command line parameters, this parameter may be omitted.

Directory

An optional string that specifies the current working directory for the program. If this parameter is omitted, the server will use the root document directory for the virtual host.

Return Value

A value of zero is returned if the program was registered successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **RegisterProgram** method registers an executable program for use with the SITE EXEC command. Because this can present a significant security risk to the server, clients are not given permission to use this command by default. A client must be explicitly granted permission to use SITE EXEC by including **ftpAccessExecute** as one of the permissions when authenticating the client session with the **Authenticate** method or creating a virtual user using the **AddUser** method.

To give the server complete control over what programs can be executed using SITE EXEC, the program must be registered with the server and referenced by an alias specified by the *CommandName* parameter. The maximum length of a program name is 31 characters and it must be at least 3 characters in length. The name must only consist of alphanumeric characters and the first character of the program name cannot be numeric. The program name is not case-sensitive, however convention is to use upper-case characters. If a program name is specified that already has been registered, it will be updated with the new information provided by this method.

The *ProgramFile* string specifies file name of the program that will be executed. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the ability to directly access the executable file itself. It is permitted to have multiple command names that reference the same executable file. The only requirement is that the command names be unique. The program name may contain environment variables surrounded by % symbols. For example, %ProgramFiles% would be expanded to the `C:\Program Files` folder.

It is important to note that the program specified by *ProgramFile* must be an executable file, not a

script or batch file. If the program name does not contain a directory path, then the standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The ***Parameters*** string is used to define optional command line parameters that will be included with the command. This string can contain placeholders that are replaced by additional parameters specified by the client when it sends the SITE EXEC command. First replacement parameter is %1, the second is %2 and so on.

The executable program that is registered using this method must be a console application that writes to standard output. Programs that write directly to a console, or programs written to use a Windows user interface are not supported and will yield unpredictable results. In most cases, those programs that do not use standard input and output will be forcibly terminated by the server. If the program attempts to read from standard input, it will immediately encounter an end-of-file condition. Programs executed by the SITE EXEC command have no input; it is similar to a program that has its input redirected from the NUL: device. If the program must process a file on the server, the local file name should be passed as a command line parameter.

The output from the program will be redirected back to the client control channel. The output should be textual, with each line of text terminated by a carriage return and linefeed (CRLF). Programs that write binary data to standard output, particular data with embedded nulls, will yield unpredictable results and are not supported. To ensure that the program output conforms to the protocol standard, any non-printable characters will be replaced with a space and each line of output will be prefixed by a single space.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

See Also

OnCommand Event, OnExecute Event

Reset Method

Reset the internal state of the control to its default values.

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. Is 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

Disconnect Method, Initialize Method, Stop Method, Uninitialize Method

ResolvePath Method

Resolve a path to its full virtual or local file name.

Syntax

object.**ResolvePath**(*ClientId*, *SourcePath*, *ResolvedPath*, [*IsVirtual*])

Parameters

ClientId

An integer that identifies the client session.

SourcePath

A string that specifies the name of the path to resolve. This may either be a virtual path, or a path to a local file name or directory.

ResolvedPath

A string that will contain the resolved path when the method returns.

IsVirtual

An optional Boolean parameter that specifies if the source path is a virtual path or local path.

Return Value

A value of zero is returned if the path could be resolved. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ResolvePath** method is used to resolve a local file name or directory to obtain its virtual path name, or obtain the full path name of a file or directory that is mapped to a virtual path. If the *IsVirtual* parameter is omitted or is **False**, the *SourcePath* parameter is considered to be a path to a local file or directory and the *ResolvedPath* parameter will contain the virtual path. If the *IsVirtual* parameter is **True**, then the *SourcePath* parameter is considered to be a virtual path and the *ResolvedPath* parameter will contain the full path to the local file or directory that the virtual path is mapped to

A virtual path for the client is either relative to the server root directory, or the client home directory if the client was authenticated as a restricted user. These virtual paths are what the client will see as an absolute path on the server. For example, if the server was configured to use "C:\ProgramData\MyServer" as the root directory, and the *SourcePath* parameter was specified as "C:\ProgramData\MyServer\Documents\Research", this method would return the virtual path to that directory as "/Documents/Research".

If the client session was authenticated as a restricted user, then the virtual path is always relative to the client home directory instead of the server root directory. This is because restricted users are isolated to their own home directory and any subdirectories. For example, if restricted user "John" has a home directory of "C:\ProgramData\MyServer\Users\John" and the *SourcePath* parameter was specified as "C:\ProgramData\MyServer\Users\John\Accounting\Projections.pdf" this method would return the virtual path as "/Accounting/Projections.pdf".

If the *SourcePath* parameter specifies a file or directory outside of the server root directory, this method will fail and the last error code will be set to **stErrorInvalidFileName**. This method can only be used with authenticated clients. If the *ClientId* parameter specifies a client session that has not been authenticated, this method will fail and the last error code will be **stErrorAuthenticationRequired**.

See Also

LocalPath Property, VirtualPath 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 server was restarted, 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. The **OnDisconnect** event will not fire for those client sessions that are terminated when the server is restarted.

See Also

Resume Method, Start Method, Stop Method, Suspend Method

Resume Method

Resume accepting new client connections.

Syntax

object.**Restart**

Parameters

None.

Return Value

A value of zero is returned if the server has resumed accepting new connections, 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 after the **Suspend** method was called.

See Also

Restart Method, Start Method, Stop Method, Suspend Method

SendResponse Method

Send a result code and message to the client in response to a command.

Syntax

object.**SendResponse**(*ClientId*, *ResultCode*, [*Message*])

Parameters

ClientId

An integer that identifies the client session.

ResultCode

An integer value that specifies the command result code to be returned to the client.

Message

An optional string value that specifies a message to be sent to the client. If this parameter is omitted is an empty string, a default message associated with the result code will be used.

Return Value

A value of zero is returned if the response was sent to the client. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **SendResponse** method is used to respond to a command issued by the client from within an **OnCommand** event handler. Command responses are normally handled by the server as a normal part of processing a command and this method is only used if the application has implemented custom commands or wishes to modify the standard responses sent by the server. The message may be a maximum of 2048 characters and may include embedded carriage-return and linefeed characters. If no message is specified, then a default message will be sent based on the result code.

Result codes must be three digits (in the range of 100 through 999) and although this method will support the use of non-standard result codes, it is recommended that the client application use the standard codes defined in RFC 959 whenever possible. The use of non-standard result codes may cause problems with FTP clients that expect specific result codes in response to a particular command.

This method should only be called once in response to a command sent by the client. If a result code has already been sent in response to a command and this method is called, it will fail and return a value of zero. This is necessary because sending multiple result codes in response to a single command may cause unpredictable behavior by the client.

See Also

OnCommand Event, OnResult Event

Start Method

Start listening for client connections on the specified IP address and port number.

Syntax

object.**Start**([*ServerAddress*], [*ServerPort*], [*Directory*] [*MaxClients*], [*IdleTime*], [*Options*])

Parameters

ServerAddress

An optional string which specifies the local hostname or IP address address that the server should be bound to. If this parameter is an empty string, then 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. If this parameter is omitted, the control will accept connections on the address specified by the value of the **ServerAddress** property.

ServerPort

An optional integer that specifies the port number the server should use to listen for client connections. If a value of zero is specified, the server will use the standard port number 21 to listen for connections, or port 990 if the server is configured to use implicit TLS. 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. If this parameter is omitted, it defaults to the value specified by the **ServerPort** property.

Directory

An optional string that specifies the path to the root directory for the server. If this parameter is omitted, it defaults to the value specified by the **Directory** property. If this property is not set and no directory is specified, the server will use the current working directory as the root directory.

MaxClients

An optional integer value that specifies the maximum number of clients that may connect to the server. If this parameter is omitted, the value specified by the **MaxClients** property will be used. This value can be adjusted after the server has been created by calling the **Throttle** method.

IdleTime

An optional integer value that specifies the number of seconds a client can be idle before the server terminates the session. If this argument is not specified, the value of the **IdleTime** property will be used. The default idle timeout period is 300 seconds (5 minutes).

Options

An optional integer value that specifies specifies one or more server options. This 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. For a list of options, see Server Option Constants.

Return Value

A value of zero is returned if the server was started, 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.

It is recommended that you always specify an absolute path for the server root directory, either by passing the full pathname as an argument to this method or by setting the **Directory** property. If the path includes environment variables surrounded by percent (%) symbols, they will be automatically expanded.

If you have configured the server to permit clients to upload files, you must ensure that your application has permission to create files in the directory that you specify. A recommended location for the server root directory would be a subdirectory of the %ALLUSERSPROFILE% directory. Using the environment variable ensures that your server will work correctly on different versions of Windows. If the root directory does not exist at the time that the server is started, it will be created.

See Also

Restart Method, Resume Method, Stop Method, Suspend Method, Throttle Method

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 server was stopped, 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.

See Also

Restart Method, Resume Method, Start Method, Suspend Method

Suspend Method

Suspend accepting new client connections.

Syntax

object.**Suspend**

Parameters

None.

Return Value

A value of zero is returned if the server has suspended accepting new connections, 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. All new clients that attempt to connect to the server will be sent a 421 "service unavailable" error code and the connection will be immediately closed. To resume accepting new client connections, call the **Resume** method. This method will not affect those clients that have already established a connection with the server before the **Suspend** method was called.

See Also

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*], [*MaxGuests*], [*ConnectionRate*])

Parameters

MaxClients

An optional integer value that specifies the maximum number of clients that may connect to the server. If this parameter is omitted, the maximum number of clients allowed will be unchanged. The default value is 100 active 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. If this parameter is omitted, the maximum number of clients per address will be unchanged. The default value is 4 client connections per address.

MaxGuests

An optional integer value that specifies the maximum number of anonymous (guest) users that may be logged in at any one time. If this parameter is omitted, the maximum number of guest users will be unchanged.

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.

If the value of the *MaxGuests* parameter is greater than zero, then anonymous logins will be enabled and clients can authenticate with the username "anonymous" and their email address as the password. If the parameter is set to zero, then anonymous logins will be disabled. Note that this will not affect any clients that are currently logged in, it only affects those clients that connect after the **Throttle**

method has been called.

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, MaxGuests Property, Resume Method, Start Method, Suspend 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. 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

File Transfer Server Control Events

Event	Description
OnAuthenticate	The client has requested authentication with the specified username and password
OnCommand	The client has issued a command to the server
OnConnect	The client established a connection to the server
OnDisconnect	The client has disconnected from the server
OnDownload	The client has downloaded a file from the server
OnError	The client encountered an error when handling a client request
OnExecute	The client has executed an external program on the server
OnIdle	The last client has disconnected from the server
OnLogin	The client has successfully authenticated the session
OnLogout	The client has logged out or reinitialized the session
OnResult	The command issued by the client has been processed by the server
OnStart	The server has started listening for connections
OnStop	The server has stopped accepting connections and all client sessions are terminated
OnTimeout	The client has exceeded the maximum allowed idle time
OnUpload	The client has uploaded a file to the server

OnAuthenticate Event

The client has requested authentication with the specified username and password.

Syntax

Sub *object*_**OnAuthenticate** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *HostName* **As Variant**, **ByVal** *UserName* **As Variant**, **ByVal** *Password* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

HostName

A string that specifies the host name that the client used to establish the connection.

UserName

A string that specifies the user name provided by the client.

Password

A string that specifies the password provided by the client.

Remarks

The **OnAuthenticate** event occurs when the client has requested authentication by sending the USER and PASS command to the server. The event handler can call the **Authenticate** method to authenticate the client session. If the client is not authenticated, the server will send an error message to the client and terminate the session.

If the application has created one or more virtual users using the **AddUser** method and/or the **LocalUser** property has been set to **True**, it is not necessary to implement an **OnAuthenticate** handler unless you also wish to perform custom authentication for specific users.

See Also

Authenticate Method, OnCommand Event, OnDownload Event, OnUpload Event

OnCommand Event

The client has issued a command to the server.

Syntax

Sub *object*_**OnCommand** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *Command* **As Variant**, **ByVal** *Parameters* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

Command

A string that specifies the command that was sent to the server.

Parameters

A string that contains any optional parameters that were sent to the server with the command. Any extraneous whitespace is removed, however quoted parameter values are unchanged.

Remarks

The **OnCommand** event occurs after the client has sent a command to the server, but before the command has been processed. This event occurs for all commands issued by the client, including invalid or disabled commands. If the application wishes to handle the command itself, it must perform any processing and then call the **SendResponse** method to send the success or error code to the client. If the **SendResponse** method is not called, then the server will perform its default processing for the command.

After the command has been processed, the **OnResult** event handler will be invoked.

See Also

CommandLine Property, SendResponse Method, OnResult Event

OnConnect Event

The client has established a connection to the server.

Syntax

Sub *object*_**OnConnect** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *ClientAddress* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

ClientAddress

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.

Remarks

The **OnConnect** event occurs after the client has established its initial connection to the server, after the server has checked the active client limits and the TLS handshake has been performed if required. If the server has been suspended, or the limit on the maximum number of client sessions has been exceeded, the server will terminate the client session prior to this event handler being invoked.

If no event handler is implemented, the server will perform the default action of accepting the connection and sending a standard greeting to the client. If you want your application to send a custom greeting to the client when it connects, call the **SendResponse** method, specifying a result code of 220 and a message of your choice.

To reject a connection, call the **SendResponse** method to send an error response to the client. Typically the result code value would be 421 to indicate that the server will not accept the connection. Next, call the **DisconnectClient** method to terminate the client session.

See Also

OnCommand Event, OnDisconnect Event

OnDisconnect Event

The client has disconnected from the server.

Syntax

Sub *object*_**OnDisconnect** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

Remarks

The **OnDisconnect** event occurs when the client disconnects from the server or when the server terminates the connection to the client by calling the **Disconnect** method. It is not required for the application to explicitly disconnect the client within the event handler, and the application cannot prevent the client from disconnecting from the server.

This event may not occur for a each client session when the server is reset or the control instance is disposed without the application first calling the **Stop** method to shutdown the server.

See Also

OnCommand Event, OnConnect Event

OnDownload Event

The client has successfully downloaded a file from the server.

Syntax

Sub *object*_**OnDownload** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *FileName* **As Variant**, **ByVal** *FileSize* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

FileName

A string that specifies the full path name of the file on the server that was downloaded.

FileSize

An integer value that specifies the number of bytes of data that was downloaded by the client.

Remarks

The **OnDownload** event occurs after the client has successfully downloaded a file from the server using the RETR command. If the file transfer fails or is aborted, this event will not occur.

See Also

OnCommand Event, OnUpload Event

OnError Event

The client encountered an error when handling a client request.

Syntax

Sub *object*_**OnError** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *ErrorCode* **As Variant**, **ByVal** *Description* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

ErrorCode

An integer value which specifies the error that has occurred.

Description

A string that describes the error.

Remarks

The **OnError** event occurs whenever the server encounters an error while accepting a client connection or processing a request. It is important to note that this event is not raised for every error that occurs. 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.
- A network error occurs when attempting to process a command issued 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.

See Also

LastError Property, LastErrorString Property, ThrowError Property

OnExecute Event

The client has executed an external program on the server.

Syntax

Sub *object*_**OnExecute** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *Program* **As Variant**, **ByVal** *Output* **As Variant**, **ByVal** *ExitCode* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

Program

A string that specifies the name of the program that was executed. This is the registered program name, and not a full path to the executable and its command arguments.

Output

A string that contains the standard output of the program that was executed. The format of this output depends on the application that was executed. If the program outputs control characters or other binary data, it will be replaced by spaces to ensure that only printable text is returned.

ExitCode

An integer value that specifies the exit code that was returned by the program.

Remarks

The **OnExecute** event occurs after the client has successfully executed an external program using the SITE EXEC command.

This event will only be generated if the client has the **ftpAccessExecute** permission. Clients are not granted this permission by default, and must be explicitly permitted to execute external programs. If the client does have this permission, it can only execute specific programs that have been registered by the server application using the **RegisterProgram** method.

See Also

RegisterProgram Method, OnCommand Event

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

OnLogin Event

The client has successfully authenticated the session.

Syntax

Sub *object*_**OnLogin** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *UserName* **As Variant**, **ByVal** *Directory* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

UserName

A string that specifies the username.

Directory

A string that specifies the full path to the home directory for the client.

Remarks

The **OnLogin** event occurs after the client has successfully authenticated itself using the USER and PASS commands.

See Also

AddUser Method, Authenticate Method, OnAuthenticate Event, OnLogout Event

OnLogout Event

The client has logged out or reinitialized the session.

Syntax

Sub *object*_**OnLogout** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *UserName* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

UserName

A string that specifies the username.

Remarks

The **OnLogout** event occurs after the client has successfully logged out using the QUIT command or reinitialized the session using the REIN command.

The application should not depend on this event handler always being invoked when a client is disconnected from the server. This event only occurs when the client sends the QUIT or REIN commands and will not be invoked if the client connection is aborted or disconnected for some other reason, such as exceeding the idle timeout period. If the application needs to update data structures or perform some cleanup when a client disconnects, that should be done in the **OnDisconnect** event handler.

The application should not call the **Disconnect** method in the handler for this event because the client is either in the process of disconnecting or expects that it can submit new credentials to the server.

See Also

OnDisconnect Event, OnLogin Event

OnResult Event

The command issued by the client has been processed by the server.

Syntax

Sub *object*_**OnResult** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *Command* **As Variant**, **ByVal** *ResultCode* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

Command

A string that specifies the command that was issued by the client.

ResultCode

An integer value that specifies the result code that was sent to the client.

Remarks

The **OnResult** event occurs after the server has processed a command issued by the client. This event will inform the application whether the command that was issued by the client was successful or not. If the command was successful, then other related events such as **OnDownload** may also fire after this event.

The ***Command*** parameter that is passed to the event handler specifies only the command itself and not any additional arguments that were included. Use the **CommandLine** property to obtain the complete command line that was issued by the client.

The ***ResultCode*** parameter is a three-digit numeric code that is used to indicate success or failure. These codes are defined as part of the File Transfer Protocol standard, with values in the range of 200-299 indicating success. Values in the range of 400-499 and 500-599 indicate failure due to various error conditions. Examples of such failures would be attempting to access a file that does not exist, issuing an unrecognized command or attempting to perform a privileged operation.

See Also

ClientDirectory Property, OnCommand 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 is generated 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 is generated 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 client has exceeded the maximum allowed idle time.

Syntax

Sub *object*_**OnTimeout** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *Elapsed* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

Elapsed

An integer value that specifies the number of seconds that have elapsed.

Remarks

The **OnTimeout** event occurs after the client has has exceeded the maximum allowed idle time, and immediately before the client is disconnected from the server. This event will never occur during a file transfer or directory listing.

To change the default idle timeout period for all clients, set the **IdleTime** property prior to starting the server. To set the idle timeout period for a specific client, set the **ClientIdle** property in an **OnConnect** or **OnLogin** event handler.

See Also

ClientIdle Property, IdleTime Property, OnConnect Event, OnLogin Event

OnUpload Event

The client has successfully uploaded a file to the server.

Syntax

Sub *object*_**OnUpload** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *FileName* **As Variant**, **ByVal** *FileSize* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

FileName

A string that specifies the full path name of the file on the server that was created or replaced.

FileSize

An integer value that specifies the number of bytes of data that was uploaded by the client.

Remarks

The **OnUpload** event occurs after the client has successfully uploaded a file to the server using the APPE, STOR or STOU command. If the file transfer fails or is aborted, this event will not occur.

See Also

OnCommand Event, OnUpload Event

Hypertext Transfer Protocol Control

Transfer files between the local system and a web server, execute scripts and perform remote file management functions.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	HttpClientCtl.HttpClient
File Name	CSHTPX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.HttpClient.11
ClassID	7ED83D99-5878-444D-80E6-01BB0939705B
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 1945, RFC 2616, RFC 7230, RFC 7540

Overview

The Hypertext Transfer Protocol (HTTP) is a lightweight, stateless application protocol that is used to access resources on web servers, as well as send data to those servers for processing. The control provides direct, low-level access to the server and the commands that are used to retrieve resources (i.e.: documents, images, etc.). The control also provides a simple interface for downloading resources to the local host, similar to how the SocketTools FTP control can be used to download files.

In a typical session, the control is used to establish a connection, send a request (to download a resource, post data for processing, etc.), read the data returned by the server and then disconnect. It is the responsibility of the client to process the data returned by the server, depending on the type of resource that was requested.

This library supports secure connections using the standard SSL and TLS protocols.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

Hypertext Transfer Protocol Control Properties

Property	Description
AuthType	Gets and sets the method used to authenticate the client session
AutoRedirect	Determines if redirected resources are handled automatically
AutoResolve	Determines if host names and IP addresses are automatically resolved
BearerToken	Gets and sets the OAuth 2.0 bearer token used for authentication
Blocking	Gets and sets the blocking state of the control
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 client certificate
CertificatePassword	Gets and sets the password associated with the client certificate
CertificateStatus	Return the status of the server certificate
CertificateStore	Gets and sets the name of the client 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	Set or return the code page used with Unicode text conversion
Compression	Set or return if data compression should be enabled
ContentLength	Return the size of the current resource in bytes
ContentType	Set or return the content type for the current resource
CookieCount	Return the number of cookies set by the server in response to a request for a resource
CookieName	Return the name of the specified cookie
CookieValue	Return the name of the specified cookie
Encoding	Gets and sets the content encoding type
FormAction	Gets and sets the path to the script that will accept the form data on the server
FormMethod	Gets and sets the method used to submit the form data
FormType	Gets and sets the type of form data encoding used to submit the form data
HashStrength	Return the length of the message digest that was selected
HeaderField	Gets and sets the name of the current header field
HeaderValue	Sets the value of a request header field or returns the value of a response header field
HostAddress	Gets and sets the IP address of the server
HostName	Gets and sets the name of the server
IsBlocked	Return if the control is blocked performing an operation
IsConnected	Determine if the control is connected to a server

IsInitialized	Determine if the control has been initialized
IsReadable	Return if data can be read from the server without blocking
IsWritable	Return if data can be sent to the server without blocking
KeepAlive	Set or return if the connection to the server is persistent
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error to occur
Localize	Determines if remote file dates are localized to the current timezone
Options	Gets and sets the options that are used in establishing a connection
Password	Gets and sets the password for the current user
Priority	Gets and sets the priority assigned to file transfers
ProtocolVersion	Gets and sets the current protocol version
ProxyHost	Gets and sets the host name of the proxy server
ProxyPassword	Gets and sets the proxy server password for the current user
ProxyPort	Gets and sets the port number for the proxy server
ProxyType	Gets and sets the current proxy server type
ProxyUser	Gets and sets the current proxy user name
RemotePort	Gets and sets the port number for a remote connection
Resource	Gets and sets the name of a resource on the HTTP server
ResultCode	Return the result code of the previous action
ResultString	Return a string describing the results of the previous action
Secure	Set or return if a connection to the server is secure
SecureCipher	Return the encryption algorithm used to establish the secure connection with the server
SecureHash	Return the message digest selected when establishing the secure connection with the server
SecureKeyExchange	Return the key exchange algorithm used to establish the secure connection with the server
SecureProtocol	Gets and sets the security protocol used to establish the secure connection with the server
TaskCount	Return the number of active background file transfers
TaskId	Return the task ID for an active background file transfer
TaskList	Return the task ID for an active background file transfer
ThrowError	Enable or disable error handling by the container of the control
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
TransferBytes	Return the number of bytes transferred from the server
TransferRate	Return the current data transfer rate in bytes per second
TransferTime	Return the number of seconds elapsed during a data transfer

URL	Gets and sets the current URL used to access a resource on the server
UserAgent	Gets and sets a value which identifies the current application
UserName	Gets and sets the current user name
Version	Return the current version of the object

AuthType Property

Gets and sets the method used to authenticate the user.

Syntax

object.**AuthType** [= *type*]

Remarks

The **AuthType** property specifies the type of authentication that should be used when the client connects to the mail server. The following authentication methods are supported:

Value	Description	
httpAuthNone	No client authentication should be performed.	
httpAuthBasic	The **Basic** authentication scheme should be used. This option is supported by all servers that support at least version 1.0 of the protocol. The user credentials are not encrypted and Basic authentication should not be used over standard (non-secure) connections. Most web services which use Basic authentication require the connection to be secure.	
4	httpAuthBearer	The **Bearer** authentication scheme should be used. This authentication method does not require a user name and the **BearerToken** property must specify the OAuth 2.0 bearer token issued by the service provider. If the access token has expired, the request will fail with an authorization error. This function will not automatically refresh an expired token.

Data Type

Integer (Int32)

Remarks

Setting the authentication type to **httpAuthNone** will remove any values set for the **UserName**, **Password** and **BearerToken** properties.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an access token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the access token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant

permission to your application to access their mail account.

Changing the value of the **BearerToken** property will automatically set the current authentication method to use OAuth 2.0.

See Also

BearerToken Property, Password Property, UserName Property, Authenticate Method, Connect Method

AutoRedirect Property

Determines if redirected resources are handled automatically by the control

Syntax

object.**AutoRedirect** [= { True | False }]

Remarks

Setting the **AutoRedirect** property determines how the control handles requests for a resource that has been moved to another location. If the property is set to True, then the control will automatically access the resource at the new location. If the property is set to False, the application is responsible for accessing the resource at its new location.

When the server indicates that a resource has been redirected, the **OnRedirect** event will fire and will provide the new location for the resource as an argument to the event handler. It is permissible for the application to change the value of the **AutoRedirect** property inside the event handler to determine whether or not the control will automatically access the resource from the new location.

Data Type

Boolean

See Also

OnRedirect Event

AutoResolve Property

Determines if host names and IP 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 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 **AutoResolve** to False.

Data Type

Boolean

See Also

HostAddress Property, HostName Property

BearerToken Property

Gets and sets the OAuth 2.0 bearer token for the current user.

Syntax

object.**BearerToken** [= *token*]

Remarks

The **BearerToken** property specifies the OAuth 2.0 bearer token used to authenticate the user. Assigning a value to this property will change the current authentication method to use OAuth 2.0 if necessary.

Your application should not store a bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

If the current authentication method does not use OAuth 2.0, this property will return an empty string and you should check the value of the **Password** property to obtain the current user's password. Refer to the **AuthType** property for more information on the available authentication methods.

Data Type

String

See Also

AuthType Property, Password Property, UserName Property, Authenticate Method, Connect 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 read data when none has been written, an error is generated. Events such as **OnConnect**, **OnDisconnect**, **OnRead** and **OnWrite** are only fired if the connection is non-blocking.

Data Type

Boolean

See Also

IsBlocked Property, IsReadable Property, IsWritable Property

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 client certificate.

Syntax

object.**CertificateName** [= *name*]

Remarks

This property sets the common name or friendly name of the certificate that should be used to establish the connection with the server. 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. If a certificate name is specified, the certificate must have a private key associated with it, otherwise the connection attempt 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 client 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
stCertificateNone	No certificate information is available. A secure connection was not established with the server.
stCertificateValid	The certificate is valid.
stCertificateNoMatch	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.
stCertificateExpired	The certificate has expired and is no longer valid. The client can examine the **CertificateExpires** property to determine when the certificate expired.
stCertificateRevoked	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.
stCertificateUntrusted	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.
stCertificateInvalid	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:

```
HttpClient1.HostName = strHostName
HttpClient1.Secure = True
```

```
nError = HttpClient1.Connect()
If nError > 0 Then
     MsgBox "Unable to connect to server " & strHostName, vbExclamation
     Exit Sub
End If

If HttpClient1.CertificateStatus <> stCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          HttpClient1.Disconnect
          Exit Sub
     End If
End If

HttpClient1.Disconnect
```

See Also

CertificateExpires Property, CertificateIssued Property, CertificateIssuer Property, CertificateSubject Property, Secure Property

CertificateStore Property

Gets and sets the name of the client certificate store or file.

Syntax

object.**CertificateStore** [= *store*]

Remarks

This property sets the name of the certificate store that contains the client certificate that should be used when establishing a secure connection with 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 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

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 client certificate.

Syntax

object.**CertificateUser** [= *username*]

Remarks

This property sets the name of the user that owns the client certificate that will be used to establish a secure connection with the server. 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 converting text to and from Unicode.

Syntax

object.**CodePage** [= *value*]

Remarks

The **CodePage** property is an integer value which specifies how text is encoded. Any valid code page identifier may be specified. Some common values are:

Value	Description
Text sent and received using a string should be converted using the ANSI code page for the current locale.	
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.	
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.	
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.	
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.	
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 exchanged with a web server 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, with each character represented using 16 bits. When you send and receive data using the String data type, they will automatically be converted to a stream of bytes.

By default, strings are converted to an array of bytes using UTF-8 encoding, mapping the 16-bit

Unicode characters to 8-bit bytes. Similarly, when reading data 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 the text you receive appears to corrupted or characters are being replaced with question marks or other symbols, it is likely the server is using a different character encoding. Most modern web services use UTF-8 encoding to represent non-ASCII characters; however, some legacy web applications may return data for its own locale rather than using Unicode. Changing this property will affect how that text is converted to Unicode.

Strings are only guaranteed to be safe when sending and receiving text. Using a string data type is not recommended when sending or receiving binary data. If possible, you should always use a byte array as the buffer parameter for the **GetData** and **PutData** methods.

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.

Although strings in Visual Basic are internally managed as Unicode, the default common controls used in Visual Basic 6.0 do not support Unicode. Those controls, such as buttons, text boxes and labels, will automatically convert the Unicode text to ANSI using the current code page. This means that text in the end-user's native language (depending on system settings) may display correctly, although text in other languages using different character sets may not. Also note that the VB6 IDE is not Unicode aware and may display corrupted string values or invalid characters, such as with tooltip values when debugging.

For Unicode support in Visual Basic 6.0, it's recommended that you use third-party controls. An alternative that some developers have used is the Microsoft Forms 2.0 Object Library (FM20.DLL) that is part of Microsoft Office. It includes a collection of controls that support Unicode, however they are not redistributable and Microsoft has stated that their use with VB6 is unsupported.

Data Type

Integer (Int32)

See Also

GetData Method, GetText Method, PostData Method, PostJson Method, PostXml Method

Compression Property

Set or return if data compression should be enabled.

Syntax

object.**Compression** [= { True | False }]

Remarks

The **Compression** property is used to indicate to the server whether or not it is acceptable to compress the data that is returned to the client. If compression is enabled, the client will advertise that it will accept compressed data by setting the **Accept-Encoding** request header. The server will decide whether a resource being requested can be compressed. If the data is compressed, the control will automatically expand the data before returning it to the caller.

Enabling compression does not guarantee that the data returned by the server will actually be compressed, it only informs the server that the client is willing to accept compressed data. Whether or not a particular resource is compressed depends on the server configuration, and the server may decide to only compress certain types of resources, such as text files. Disabling compression informs the server that the client is not willing to accept compressed data; this is the default.

If the **SetHeader** method is used to explicitly set the **Accept-Encoding** header to request compressed data and compression is not enabled, the control will not attempt to automatically expand the data returned by the server. In this case, the raw compressed data will be returned and the application is responsible for processing it. This behavior is by design to maintain backwards compatibility with previous versions of the control that did not have internal support for compression.

To determine if the server compressed the data returned to the client, use the **GetHeader** method to get the value of the **Content-Encoding** header. If the header is defined, the value specifies the compression method used, otherwise the data was not compressed.

This property value is only meaningful when downloading files from a server that supports file compression. It has no effect on file uploads.

Data Type

Boolean

See Also

GetData Method, GetFile Method, GetHeader Method SetHeader Method

ContentLength Property

Return the content length for the current resource.

Syntax

object.**ContentLength**

Remarks

Remarks

The **ContentLength** property is a read-only value that returns the size of the current resource which has been retrieved from the server. When this property value is checked after a GET or POST request has completed, the value will be the total number of bytes returned by the server. If this property is checked while a request is being processed (for example, from within an event handler), it will return the value of the Content-Length response header provided by the server.

Most servers will only include a Content-Length header in the response for static resources like documents or image files. For other types of resources where the response payload is generated dynamically, such as server-side scripts, servers typically use "chunked" output and there is no Content-Length header. The server may also not provide a file size for HTML documents which use server side includes (SSI) because that content is also dynamically created by the server. In this case, the content length can only be determined after the request has completed.

This property value should only be checked after a request has completed and will not return a value for dynamic resources as they are being retrieved from the server. To obtain the current number of bytes copied from within an event handler, check the value of the **TransferBytes** property.

If the content type for the current resource is textual, the content length reported by the server may be different than what is stored on the local system due to different end-of-line character conventions. For example, on Windows the end of a line of text is indicated by a carriage-return and linefeed pair, while on Linux the end-of-line is indicated by a single linefeed. Depending on the options specified when the request is made, text returned by the server may be normalized for the Windows platform.

This property does not use the HEAD command to request the size of a document on the server. It will only return the content length of a resource which has already been requested, typically after using methods such as **GetText** or **PostXml**. To ask the server for the size of document without downloading the contents of that document, use the **GetFileSize** method.

Data Type

Integer (Int32)

See Also

ContentLengthXL Property, ContentType Property, TransferBytes Property, GetFileSize Method, GetHeader Method, SetHeader Method

ContentLengthXL Property

Return the content length for the current resource.

Syntax

object.**ContentLengthXL**

Remarks

Remarks

The **ContentLengthXL** property is a read-only value that returns the size of the current resource which has been retrieved from the server. When this property value is checked after a GET or POST request has completed, the value will be the total number of bytes returned by the server. If this property is checked while a request is being processed (for example, from within an event handler), it will return the value of the Content-Length response header provided by the server. This property returns the number of bytes as a **Double** floating-point value instead of a **Long** integer, making it suitable for very large files that exceed 2 GiB in size.

Most servers will only include a Content-Length header in the response for static resources like documents or image files. For other types of resources where the response payload is generated dynamically, such as server-side scripts, servers typically use "chunked" output and there is no Content-Length header. The server may also not provide a file size for HTML documents which use server side includes (SSI) because that content is also dynamically created by the server. In this case, the content length can only be determined after the request has completed.

This property value should only be checked after a request has completed and will not return a value for dynamic resources as they are being retrieved from the server. To obtain the current number of bytes copied from within an event handler, check the value of the **TransferBytesXL** property.

If the content type for the current resource is textual, the content length reported by the server may be different than what is stored on the local system due to different end-of-line character conventions. For example, on Windows the end of a line of text is indicated by a carriage-return and linefeed pair, while on Linux the end-of-line is indicated by a single linefeed. Depending on the options specified when the request is made, text returned by the server may be normalized for the Windows platform.

This property does not use the HEAD command to request the size of a document on the server. It will only return the content length of a resource which has already been requested, typically after using methods such as **GetText** or **PostXml**. To ask the server for the size of document without downloading the contents of that document, use the **GetFileSize** method.

Data Type

Double

See Also

ContentLength Property, ContentType Property, TransferBytesXL Property, GetFileSize Method, GetHeader Method, SetHeader Method

ContentType Property

Set or return the content type for the current resource.

Syntax

object.**ContentType** [= *value*]

Remarks

The **ContentType** property returns content type for the current resource. Changing this value sets the content type for the next request submitted to the server.

This property returns a value based on the Content-Type response header and does not examine the contents of the payload returned by the server. If the web server does not recognize the data format of the resource it is returning, this property should return a value of **application/octet-stream**. However, some servers incorrectly return unrecognized formats as **text/plain**, causing the payload to be identified as human-readable text rather than binary data.

Some servers will return a content type of **text/plain** for JSON responses and others will use the IANA standard type of **application/json**. If the server returns an XML payload, it may indicate the content type either as **text/xml** or **application/xml**.

Data Type

String

See Also

HeaderField Property, HeaderValue Property, GetHeader Method, SetHeader Method

CookieCount Property

Return the number of cookies set by the server in response to a request for a resource.

Syntax

object.**CookieCount**

Remarks

The **CookieCount** property returns the number of cookies that were set by the server. This value can be used in conjunction with the **CookieName** and **CookieValue** properties to enumerate all of the available cookies and their values.

Data Type

String

Example

```
' Save the cookies set by a previous request to this server for
' a resource so that they can be sent back with the next request
nCookies = HttpClient1.CookieCount
ReDim strCookieName(nCookies)
ReDim strCookieValue(nCookies)

' Enumerate the available cookies and store them in the array;
' a more complex implementation could use the GetCookie method
' to check for additional information about the cookie, such
' as whether the cookie should be stored locally on the system
For nIndex = 0 To nCookies - 1
  strCookieName(nIndex) = HttpClient1.CookieName(nIndex)
  strCookieValue(nIndex) = HttpClient1.CookieValue(nIndex)
Next

' Clear any previously set headers
HttpClient1.ClearHeaders

' Set each of the cookies that were stored in the array
For nIndex = 0 To nCookies - 1
  HttpClient1.SetCookie strCookieName(nIndex), strCookieValue(nIndex)
Next

' Request the next resource from the server and store
' the data in the strResult string buffer
nError = HttpClient1.GetData(strResource, strResult)
```

See Also

CookieName Property, CookieValue Property, GetCookie Method, SetCookie Method

CookieName Property

Return the name of the specified cookie.

Syntax

object.**CookieName**(*Index*)

Remarks

The **CookieName** property array returns a string which identifies the cookie specified by the *Index* argument. The array is zero based, which means the name of the first available cookie is read by using an index value of zero. The **CookieCount** property indicates the total number of cookies that have been returned by the server.

Data Type

String

Example

```
' Save the cookies set by a previous request to this server for
' a resource so that they can be sent back with the next request
nCookies = HttpClient1.CookieCount
ReDim strCookieName(nCookies)
ReDim strCookieValue(nCookies)

' Enumerate the available cookies and store them in the array;
' a more complex implementation could use the GetCookie method
' to check for additional information about the cookie, such
' as whether the cookie should be stored locally on the system
For nIndex = 0 To nCookies - 1
  strCookieName(nIndex) = HttpClient1.CookieName(nIndex)
  strCookieValue(nIndex) = HttpClient1.CookieValue(nIndex)
Next

' Clear any previously set headers
HttpClient1.ClearHeaders

' Set each of the cookies that were stored in the array
For nIndex = 0 To nCookies - 1
  HttpClient1.SetCookie strCookieName(nIndex), strCookieValue(nIndex)
Next

' Request the next resource from the server and store
' the data in the strResult string buffer
nError = HttpClient1.GetData(strResource, strResult)
```

See Also

CookieCount Property, CookieValue Property, GetCookie Method, SetCookie Method

CookieValue Property

Return the name of the specified cookie.

Syntax

object.**CookieValue**(*Index*)

Remarks

The **CookieValue** property array returns a string which contains the value for the cookie specified by the *Index* argument. The array is zero based, which means the value of the first available cookie is read by using an index value of zero. The **CookieCount** property indicates the total number of cookies that have been returned by the server.

Data Type

String

Example

```
' Save the cookies set by a previous request to this server for
' a resource so that they can be sent back with the next request
nCookies = HttpClient1.CookieCount
ReDim strCookieName(nCookies)
ReDim strCookieValue(nCookies)

' Enumerate the available cookies and store them in the array;
' a more complex implementation could use the GetCookie method
' to check for additional information about the cookie, such
' as whether the cookie should be stored locally on the system
For nIndex = 0 To nCookies - 1
  strCookieName(nIndex) = HttpClient1.CookieName(nIndex)
  strCookieValue(nIndex) = HttpClient1.CookieValue(nIndex)
Next

' Clear any previously set headers
HttpClient1.ClearHeaders

' Set each of the cookies that were stored in the array
For nIndex = 0 To nCookies - 1
  HttpClient1.SetCookie strCookieName(nIndex), strCookieValue(nIndex)
Next

' Request the next resource from the server and store
' the data in the strResult string buffer
nError = HttpClient1.GetData(strResource, strResult)
```

See Also

CookieCount Property, CookieName Property, GetCookie Method, SetCookie Method

Encoding Property

Gets and sets the content encoding type.

Syntax

object.**Encoding** = [*type*]

Remarks

The **Encoding** property explicitly sets the type of encoding used when optional parameter data is submitted with a request for a resource. By default, data is URL encoded and the content type will be designated as application/x-www-form-urlencoded. The following encoding types are supported:

Value	Description
httpEncodingNone	No encoding will be applied to the content of a request and no default content type will be specified. This encoding type should be used with REST APIs and other services which expect XML or JSON request payloads.
httpEncodingURL	Non-printable and extended ASCII characters will be encoded so they can be safely used with URLs and form data. Encoded characters will be represented by a percent symbol prefix, followed by a two digit hexadecimal value which represents the ASCII character code. This encoding is typically used with web services which process HTML form data.
httpEncodingXML	This encoding is identical to URL encoding, except spaces are not encoded. It is used with legacy web services which expect form data in an XML format and cannot process encoded whitespace. This encoding should not be specified for services which use REST APIs.

The **Encoding** property explicitly sets the type of encoding used when optional parameter data is submitted with a request for a resource. If an encoding type is specified, and the content type for the request payload has not been defined, it will default to **application/x-www-form-urlencoded**.

When submitting a JSON or XML request to a service using a REST API, your application should use **httpEncodingNone** and set the appropriate content type for the request payload. The **httpEncodingXml** encoding type should only be used if the server expects URL encoded form data. The **PostJson** and **PostXml** methods will automatically set the correct encoding and content type for those requests.

If an application specifies **httpEncodingNone**, parameter data is not encoded and no content type header will created by default. The client application can specify the content type by calling the **SetHeader** method.

Data Type

Integer (Int32)

See Also

HeaderField Property, HeaderValue Property, PostData Method, PostJson Method, PostXml Method, SetHeader Method

FormAction Property

Gets and sets the path to the script that will accept the form data on the server.

Syntax

object.**FormAction** [= *value*]

Remarks

The **FormAction** property is used to specify the name of the script that will process the form data submitted by the control. This property is only used by the **SubmitForm** method and changing the property value does not change the current resource.

Data Type

String

See Also

FormMethod Property, FormType Property, CreateForm Method, SubmitForm Method

FormMethod Property

Gets and sets the method used to submit the form data.

Syntax

object.**FormMethod** [= *value*]

Remarks

The **FormMethod** property is used to specify how form data will be submitted to the server using the **SubmitForm** method. It may be one of the following values:

Value	Description
httpMethodGet	The form data should be submitted using the GET command. This method should be used when the amount of form data is relatively small. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used instead.
httpMethodPost	The form data should be submitted using the POST command. This is the preferred method of submitting larger amounts of form data. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used.

Data Type

Integer (Int32)

See Also

FormAction Property, FormType Property, CreateForm Method, SubmitForm Method

FormType Property

Gets and sets the type of form data encoding used to submit the form data.

Syntax

object.**FormType** [= *value*]

Remarks

The **FormType** property is used to specify how form data will be encoded when it is submitted to the server using the **SubmitForm** method. It may be one of the following values:

Value	Description
httpFormEncoded	The form data should be submitted as URL encoded values. This is typically used when the GET method is used to submit the data to the server.
httpFormMultipart	The form data should be submitted as multipart form data. This is typically used when the POST method is used to submit a file to the server. Note that the script must understand how to process multipart form data if this form type is specified.

Data Type

Integer (Int32)

See Also

FormAction Property, FormMethod Property, CreateForm Method, SubmitForm 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

HeaderField Property

Gets and sets the name of the current header field.

Syntax

object.**HeaderField** [= *value*]

Remarks

The **HeaderField** property is used in conjunction with the **HeaderValue** property to set and/or get the values of specific fields in the HTTP request header. For example, setting this property to the value "Content-Length" and then reading the value of the HeaderValue property would cause the control to return length (in bytes) of the specified resource.

Note that the control automatically generates a default request header, and it is not required that the client use the **HeaderField** and **HeaderValue** properties unless it has a specific need to do so.

Data Type

String

See Also

CookieCount Property, CookieName Property, CookieValue Property, HeaderValue Property, GetCookie Method, GetHeader Method, SetCookie Method, SetHeader Method

HeaderValue Property

Sets the value of a request header field or returns the value of a response header field.

Syntax

object.**HeaderValue** [= *value*]

Remarks

The **HeaderValue** property is used in conjunction with the **HeaderField** property to set or get the values of specific fields in the HTTP request and response headers. When the property is set to a value, then the specified request header field is set to this value. When the property is read, then the value associated with the specified response header field is returned.

Note that the control automatically generates a default request header, and it is not required that the client use the **HeaderField** and **HeaderValue** properties unless it has a specific need to do so.

Data Type

String

See Also

CookieCount Property, CookieName Property, CookieValue Property, HeaderField Property, GetCookie Method, GetHeader Method, SetCookie Method, SetHeader Method

HostAddress Property

Gets and sets the IP address of the server.

Syntax

object.**HostAddress** [= *ipaddress*]

Remarks

The **HostAddress** property can be used to set the IP address for a server 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, HostName Property, URL Property

HostName Property

Gets and sets the name of the server.

Syntax

object.**HostName** [= *hostname*]

Remarks

The **HostName** property should be set to the name of the server 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, URL Property

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.

Note that this property will return True if there is ***any*** blocking operation being performed by the application, regardless if the specified control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

Blocking Property, LastError Property

IsConnected Property

Determine if the control is connected to a server.

Syntax

object.**IsConnected**

Remarks

The **IsConnected** read-only property is set to a value of true if the control is connected with a server, otherwise the property has a value of false.

Data Type

Boolean

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

IsReadable Property

Return if data can be read from the server without blocking.

Syntax

object.**IsReadable**

Remarks

The **IsReadable** property returns True if data can be read from the server without blocking. For non-blocking connections, this property can be checked before the application attempts to read the data, preventing an error.

Data Type

Boolean

See Also

IsConnected Property, Read Method, OnRead Event

IsWritable Property

Return if data can be sent to the server without blocking.

Syntax

object.**IsWritable**

Remarks

The **IsWritable** property returns True if data can be written without blocking. For non-blocking connections, this property can be checked before the application attempts to send data to the server, 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 send data without an error. The next operation may result in an **stErrorOperationWouldBlock** or **stErrorOperationInProgress** error. The application must treat these errors as recoverable, and should be prepared to retry operations that result in them.

Data Type

Boolean

See Also

IsReadable Property, Write Method, OnWrite Event

KeepAlive Property

Set or return if the connection to the server is persistent

Syntax

object.**KeepAlive** [= { True | False }]

Remarks

Setting the **KeepAlive** property to a value of true indicates that the client wishes to maintain a persistent connection with the server. For those clients who wish to retrieve a number of documents, this is more efficient because the client does not need to connect, retrieve the document and disconnect each time. Instead, the client can connect, retrieve each document and then disconnect when it is finished. If the property value is False, a persistent connection is not maintained, and the client must establish a connection for each document that it wishes to retrieve. This property should be set to the desired value before establishing a connection with the server.

Note that this option is only available for those servers which support version 1.0 or later of the HTTP protocol. For version 1.0 servers, the connection header field is set to the value 'keep-alive', which instructs compliant servers to maintain a persistent connection. For version 1.1 and later, persistent connections are the default. In this case, if the property value is set to False, the connection header field will be set to the value 'close', telling the server that you wish to close the connection after the document has been retrieved. It is possible that the server may choose to close the connection itself, even if it supports persistent connections. If the server does not support persistent connections and the **KeepAlive** property is set to True, the control will attempt to simulate them by automatically reconnecting for each request.

Data Type

Boolean

See Also

ProtocolVersion Property, Connect Method, GetFile 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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

Localize Property

Determines if remote file dates are localized to the current timezone.

Syntax

object.**Localize** [= { True | False }]

Remarks

Setting the **Localize** property controls how remote file date and time values are localized when the **GetFileTime** method is called. If the property is set to True, then the file date and time will be adjusted to the current timezone. If the property is set to False, which is the default value, then the file date and time are returned as UTC (Coordinated Universal Time) values.

Data Type

Boolean

See Also

GetFileTime Method

Options Property

Gets and sets the options that are used in establishing a connection.

Syntax

object.**Options** [= *value*]

Remarks

The **Options** property is an integer value which specifies one or more options. The value specified for this property will be used as the default options when connecting to the server. The property value is created by using a bitwise operator with one or more of the following values

Value	Description
httpOptionNoCache	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
httpOptionKeepAlive	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate connection for each resource that is requested.
httpOptionRedirect	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
httpOptionProxy	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection

	will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed in the system settings or control panel.	
httpOptionErrorData	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with **httpOptionRedirect**.	
httpOptionNoUserAgent	This option specifies the client should not include a User-Agent header with any requests made during the session. The user agent is a string which is used to identify the client application to the server. An application can provide its own custom user agent value using the **SetHeader** method.	
httpOptionHttp2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol. The value of the **ProtocolVersion** property will be ignored when this option is used.	
&H400	httpOptionTunnel	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior

		of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
&H800	httpOptionTrustedSite	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 the TLS protocol.
&H1000	httpOptionSecure	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol. The client will default to using TLS 1.2 or later for secure connections.
&H8000	httpOptionSecureFallback	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.
&H40000	httpOptionPreferIPv6	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.
&H100000	httpOptionHiResTimer	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller

		amounts of data over fast network connections.

Remarks

If the **httpOptionKeepAlive** option is specified and the server does not support persistent connections, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers. This option is automatically enabled when using HTTP/2.

If your application specifies the **httpOptionHttp2** option, a secure connection using TLS 1.2 or later will always be used. The minimum required platform for HTTP/2 support is Windows 10 (Build 1903) or Windows Server 2019. Earlier versions of Windows do not support the features required for a secure HTTP/2 connection. If the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1.

Data Type

Integer (Int32)

See Also

AutoRedirect Property, KeepAlive Property, Secure Property, Connect Method

Password Property

Gets and sets the password for the current user.

Syntax

object.**Password** [= *password*]

Remarks

The **Password** property specifies the password used to authenticate the user on the current server. This property must be set to access resources that are restricted on the server. Note that it is required to set both the **UserName** and **Password** properties to enable client authentication.

If your application needs to use OAuth 2.0 for authentication, it is recommended you set the **BearerToken** property, which will automatically set the correct authentication type.

Data Type

String

See Also

AuthType Property, BearerToken Property, UserName Property, Authenticate Method, Connect Method

Priority Property

Gets and sets a value which specifies the priority of file transfers.

Syntax

object.**Priority** [= *priority*]

Remarks

The **Priority** property can be used to control the processor usage, memory and network bandwidth allocated for file transfers. One of the following values may be specified:

Value	Description
httpPriorityBackground	This priority significantly reduces the memory, processor and network resource utilization for the transfer. It is typically used with worker threads running in the background when the amount of time required perform the transfer is not critical.
httpPriorityLow	This priority lowers the overall resource utilization for the transfer and meters the bandwidth allocated for the transfer. This priority will increase the average amount of time required to complete a file transfer.
httpPriorityNormal	The default priority which balances resource utilization and transfer speed. It is recommended that most applications use this priority.
httpPriorityHigh	This priority increases the overall resource utilization for the transfer, allocating more memory for internal buffering. It can be used when it is important to transfer the file quickly, and there are no other threads currently performing file transfers at the time.
httpPriorityCritical	This priority can significantly increase processor, memory and network utilization while attempting to transfer the file as quickly as possible. If the file transfer is being performed in the main UI thread, this priority can cause the application to appear to become non-responsive. No events will be generated during the transfer.

The **httpPriorityNormal** priority balances resource utilization and transfer speed while ensuring that a single-threaded application remains responsive to the user. Lower priorities reduce the overall resource utilization at the expense of transfer speed. For example, if you create a worker thread to download a file in the background and want to ensure that it has a minimal impact on the process, the **httpPriorityBackground** value can be used.

Higher priority values increase the memory allocated for the transfers and increases processor utilization for the transfer. The **httpPriorityCritical** priority maximizes transfer speed at the expense of system resources. It is not recommended that you increase the file transfer priority unless you understand the implications of doing so and have thoroughly tested your application. If the file transfer is being performed in the main UI thread, increasing the priority may interfere with the normal processing of Windows messages and cause the application to appear to become non-responsive. It is also important to note that when the priority is set to **httpPriorityCritical**, normal progress events will not be generated during the transfer.

Data Type

Integer (Int32)

See Also

GetData Method GetFile Method PutData Method PutFile Method

ProtocolVersion Property

Gets and sets the current protocol version.

Syntax

object.**ProtocolVersion** [= *value*]

Remarks

The **ProtocolVersion** property sets or returns the current HTTP version number. It is used to determine how requests are submitted to the server, as well as what header fields are required. The default value for this property is "1.1", and should be changed before any connection attempt is made by the client. It is recommended you use this default value to ensure the broadest compatibility with most servers and Windows platforms.

Setting this property to a value of "2.0" specifies the client should use the HTTP/2 protocol standard defined in RFC 7540. This protocol version is a significant change from previous versions and can provide improved performance with header compression and optimizing how requests are serviced. However, this version should only be used if the server supports HTTP/2 and the client is running on Windows 10 (Build 1903) or Windows Server 2019 or later versions. Earlier versions of the Schannel SSP do not support the features required for a secure HTTP/2 connection.

Data Type

String

See Also

KeepAlive Property, Connect Method

ProxyHost Property

Gets and sets the host name of the proxy server.

Syntax

object.**ProxyHost** [= *hostname*]

Remarks

The **ProxyHost** property should be set to the name of the proxy server that you want to connect to. This property may be set to either a fully qualified domain name, or an IP address. This property is only used if the **ProxyType** property is set to a non-zero value.

Data Type

String

See Also

ProxyPassword Property, ProxyPort Property, ProxyType Property, ProxyUser Property, Connect Method

ProxyPassword Property

Gets and sets the proxy server password for the current user.

Syntax

object.**ProxyPassword** [= *password*]

Remarks

The **ProxyPassword** property specifies the password used to authenticate the user to the proxy server. If a password is not required by the server, this property is ignored.

Data Type

String

See Also

ProxyHost Property, ProxyPort Property, ProxyType Property, ProxyUser Property, Connect Method

ProxyPort Property

Gets and sets the port number for the proxy server.

Syntax

object.**ProxyPort** [= *portnumber*]

Remarks

The **ProxyPort** property is used to set the port number that the control will use to establish a connection with the proxy server. A value of zero specifies that the client will connect to the proxy server using the standard HTTP service port.

Data Type

Integer (Int32)

See Also

ProxyHost Property, ProxyPassword Property, ProxyType Property, ProxyUser Property, Connect Method

ProxyType Property

Gets and sets the current proxy server type.

Syntax

object.**ProxyType** = [*proxytype*]

Remarks

The **ProxyType** property specifies the type of proxy server that the client is connecting to. The supported proxy server types are as follows:

Value	Description
httpProxyNone	A direct connection will be established with the server. When this value is specified the proxy-related properties are ignored.
httpProxyStandard	A standard connection is established through the specified proxy server, and all resource requests will be specified using a complete URL. This proxy type should be used with standard connections.
httpProxySecure	A secure connection is established through the specified proxy server. This proxy type should only be used with secure connections and the Secure property should also be set to a value of true.
httpProxyWindows	The configuration options for the current system should be used. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. These settings are the same proxy server settings configured in Windows.

If the **httpProxyWindows** proxy type is specified, then the proxy configuration for the local system is used. If no proxy server has been defined, then the proxy-related properties will be ignored and the **Connect** method will establish a connection directly to the server.

Data Type

Integer (Int32)

See Also

ProxyHost Property, ProxyPassword Property, ProxyPort Property, ProxyUser Property, Secure Property, Connect Method

ProxyUser Property

Gets and sets the current proxy user name.

Syntax

object.**ProxyUser** [= *username*]

Remarks

The **ProxyUser** property specifies the user that is logging in to the proxy server. If the proxy server does not require user authentication, then this property is ignored.

Data Type

String

See Also

ProxyHost Property, ProxyPassword Property, ProxyPort Property, ProxyType Property, Connect Method

RemotePort Property

Gets and sets the port number for a remote connection.

Syntax

object.**RemotePort** [= *portnumber*]

Remarks

The **RemotePort** property is used to set the port number that the control will use to establish a connection with the server.

Data Type

Integer (Int32)

See Also

HostAddress Property, HostName Property, URL Property

Resource Property

Gets and sets the name of a resource on the HTTP server.

Syntax

object.**Resource** [= *value*]

Remarks

The **Resource** property is used to specify the name of a resource on the server. The resource may be a file, such as an HTML document or an image, or it may be a script used to process data submitted by the client. Note that this property specifies the name of the resource only, not a complete URL. To specify a complete URL, set the **URL** property and the control will automatically set the **Resource** property to the correct value.

Data Type

String

See Also

URL Property, Connect Method, GetData Method, GetFile Method, PostData Method

ResultCode Property

Return the result code of the previous action.

Syntax

object.**ResultCode**

Remarks

The **ResultCode** read-only property returns the result code of the last action performed by the client. This property should be checked after the **Command** method is used to execute a command on the server to determine if the operation was successful. Result codes are three-digit numeric values returned by the server and may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

It is important to note that while some result codes have become standardized, not all servers respond to commands using the same result codes. For example, one server may respond with a result code of 221 to indicate success, while another may respond with a value of 235. It is recommended that applications check for ranges of values to determine if a command was successful, not a specific value.

Data Type

Integer (Int32)

See Also

ResultString Property, Command Method, OnCommand Event

ResultString Property

Return a string describing the results of the previous action.

Syntax

object.**ResultString**

Remarks

The **ResultString** read-only property returns the result string from the last action taken by the client. This string is generated by the server, and typically is used to describe the result code. For example, if an error is indicated by the result code, the result string may describe the condition that caused the error.

Data Type

String

See Also

ResultCode Property, Command Method, OnCommand Event

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 must 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 strongly recommended that any application that sets this property True use error handling to trap an errors that may occur. If the control 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.

Data Type

Boolean

Example

The following example establishes a secure connection to a server and retrieves a file:

```
HttpClient1.HostName = strHostName
HttpClient1.RemotePort = 443
HttpClient1.Secure = True

nError = HttpClient1.Connect()
If nError > 0 Then
    MsgBox "Unable to connect to server " & strHostName, vbExclamation
    Exit Sub
End If

If HttpClient1.CertificateStatus <> stCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          HttpClient1.Disconnect
          Exit Sub
     End If
End If

nError = HttpClient1.GetFile(strLocalFile, strRemoteFile)
HttpClient1.Disconnect

If nError > 0 Then
    MsgBox "Unable to download " & strRemoteFile, vbExclamation
    Exit Sub
End If
```

See Also

CertificateStatus 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
stCipherNone	No cipher has been selected. This is not a secure connection with the server.
stCipherRC2	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.
stCipherRC4	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.
stCipherRC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
stCipherDES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher using 56-bit keys.
stCipherDES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively using a 168-bit key length.
stCipherDESX	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.
stCipherAES	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.
stCipherSkipjack	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
stCipherBlowfish	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
stHashMD5	The MD5 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA1	The SHA-1 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA256	The SHA-256 algorithm has been selected.
stHashSHA384	The SHA-384 algorithm has been selected.
stHashSHA512	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
stKeyExchangeNone	No key exchange algorithm has been selected. This is not a secure connection with the server.
stKeyExchangeRSA	The RSA public key exchange algorithm has been selected.
stKeyExchangeKEA	The KEA public key exchange algorithm has been selected. This is an improved version of the Diffie-Hellman public key algorithm.
stKeyExchangeDH	The Diffie-Hellman public key exchange algorithm has been selected.
stKeyExchangeECDH	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

TaskCount Property

Return the number of active background file transfers.

Syntax

object.**TaskCount**

Remarks

The **TaskCount** property returns the number of background file transfers that are currently in progress. One common use for this property is to create a timer that periodically checks this value when a series of background transfers are started. When the property returns a value of zero, that indicates all of the background transfers have completed. This property can also be used to enumerate the active background tasks in conjunction with the **TaskList** property.

Data Type

Integer (Int32)

See Also

TaskList Property, AsyncGetFile Method, AsyncPutFile Method, TaskAbort Method, TaskWait Method

TaskId Property

Return the task ID for the last background file transfer.

Syntax

object.**TaskId**

Remarks

The **TaskId** property returns the task ID associated with the last background task that started. The value of this property is only meaningful after the **AsyncGetFile** or **AsyncPutFile** method is called to initiate a background file transfer, and the value will change with each subsequent background transfer that is performed. If this property returns a value of zero, that indicates that no background tasks have been started for this instance of the control.

To enumerate the active background tasks, use the **TaskCount** property and the **TaskList** property array.

Data Type

Integer (Int32)

See Also

TaskCount Property, TaskList Property, AsyncGetFile Method, AsyncPutFile Method, TaskAbort Method, TaskWait Method

TaskList Property

Return the task ID for an active background file transfer.

Syntax

object.**TaskList**(*Index*)

Remarks

The **TaskList** property is a zero-based array that returns an ID associated with an active background task. The current number of active tasks can be determined using the **TaskCount** property. If the index value specified for this property array exceeds the number of active tasks, an exception will be thrown.

As background tasks complete and additional tasks are started, the values returned by this property array will change. The application should never make an assumption about the actual task ID values returned or the order they are returned. While task IDs are assigned sequentially, they should be considered opaque values that are unique to the process. When a background task completes, its corresponding task ID is removed from the list of active tasks and this can potentially change the task ID values associated with each index into the property array.

Data Type

Integer (Int32)

See Also

TaskCount Property, TaskId Property, AsyncGetFile Method, AsyncPutFile Method, TaskAbort Method, TaskWait Method

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
HttpClient1.ThrowError = False
nError = HttpClient1.Connect(strHostName)

If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

HttpClient1.ThrowError = True
HttpClient1.Connect strHostName

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error.

Note that the **Timeout** property also determines the amount of time the control will spend attempting to connect to a server. If a connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

When debug logging is enabled for HTTP/2 client sessions, it is normal to see HTTP/1.1 in the request and server response. HTTP/2 is a binary protocol and the request and response header blocks emulate a standard HTTP/1.1 text response for backwards compatibility. Applications running on the server should work in the same way regardless of which protocol version is selected, however it is possible to check the server environment to determine which version of HTTP was used with the request.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
httpTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
httpTraceError	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.
httpTraceWarning	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.
httpTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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

TransferBytes Property

Return the number of bytes transferred from the server.

Syntax

object.**TransferBytes**

Remarks

The **TransferBytes** property returns the number of bytes that have been copied to or from the HTTP server. If this property is read while a transfer is ongoing, the property returns the number of bytes that have been copied up to that point. If read after a transfer has completed, the total number of bytes copied is returned.

If the value would exceed 2,147,483,647 bytes (the maximum value for a 32-bit integer) this property will return -1 to indicate an overflow condition. If you are potentially transferring files larger than 2 GiB in size, you should use the **TransferBytesXL** property instead, which returns the number of bytes as a **Double** floating-point value.

This property value is reset with every data transfer.

Data Type

Integer (Int32)

See Also

ContentLength Property, ContentType Property, TransferBytesXL Property, TransferRate Property, TransferTime Property, OnProgress Event

TransferBytesXL Property

Return the number of bytes transferred from the server.

Syntax

object.**TransferBytesXL**

Remarks

The **TransferBytesXL** property returns the number of bytes that have been copied to or from the HTTP server. This property returns the number of bytes as a **Double** floating-point value instead of a **Long** integer, making it suitable for very large files that exceed 2 GiB in size.

If this property is read while a transfer is ongoing, the property returns the number of bytes that have been copied up to that point. If read after a transfer has completed, the total number of bytes copied is returned.

This property value is reset with every data transfer.

Data Type

Double

See Also

ContentLength Property, ContentType Property, TransferRate Property, TransferTime Property, OnProgress Event

TransferRate Property

Return the current file transfer rate in bytes per second.

Syntax

object.**TransferRate**

Remarks

The **TransferRate** property returns the rate at which the file data is being transferred, expressed in bytes per second. If this property is read while a transfer is ongoing, it returns the current average transfer rate.

If this property is read after the transfer has completed, it returns the final transfer rate which is calculated as the total number of bytes transferred divided by the number of seconds to complete the transfer. This property value is reset with every data transfer.

Data Type

Integer (Int32)

See Also

TransferBytes Property, TransferTime Property, OnProgress Event

TransferTime Property

Return the number of seconds elapsed during a data transfer.

Syntax

object.**TransferTime**

Remarks

The **TransferTime** property returns the number of seconds that have elapsed since the file transfer started. If the property is read after the transfer has completed, it returns the total number of seconds it took to transfer the file.

This property value is reset with every data transfer.

Data Type

Integer (Int32)

See Also

TransferBytes Property, TransferRate Property, OnProgress Event

URL Property

Gets and sets the current URL used to access a resource on the server.

Syntax

object.**URL** [= *url*]

Remarks

The **URL** property returns the current Uniform Resource Locator string which is used by the control to access a resource on the server. URLs have a specific format which provides information about the server, port, resource, as well as optional information such as a username and password for authentication:

```
http://[username : [password] @] hostname [:port] / resource [? parameters]
```

The first part of the URL is the protocol and in this case will always be "http", or "https" if a secure connection is being used. If a username and password is required for authentication, then this will be included in the URL before the name of the server. Next, there is the name of the server to connect to, optionally followed by a port number. If no port number is given, then the default port for the protocol will be used. This is followed by the resource, which is usually a path to a file or script on the server. Parameters to the resource may also be specified, which are typically used as arguments to a script that is executed on the server.

Here are some common examples of URLs used to access resources on an HTTP server:

`http://www.example.com/products/index.html`
In this example, the server is www.example.com and the resource is /products/index.html. The default port will be used to access the resource, and no username and password is provided for authentication.

`http://www.example.com:8080/index.html`
In this example, the server is www.example.com and the resource is /products/index.html. However, the client should connect to an alternative port number, in this case 8080.

`https://www.example.com/order/confirm.asp`
In this example, the server is www.example.com and the resource is the script /order/confirm.asp. Because the protocol is https, a secure connection on port 443 will be established.

`http://jsmith:secret@www.example.com:8080/~jsmith/personal/index.html`
In this example, the server is www.example.com and the resource is /~jsmith/personal/index.html. The port 8080 will be used to access the resource, and access to the resource will be authenticated with the username "jsmith" and the password "secret".

When setting the **URL** property, the control will parse the string and automatically update the **HostName**, **RemotePort**, **UserName**, **Password** and **Resource** properties according to the values specified in the URL. This enables an application to simply provide the URL and then call the **Connect** method to establish the connection.

Note that if this property is assigned a value which cannot be parsed, the control will throw an error that indicates that the property value is invalid. In a language like Visual Basic it is important that you implement an error handler, particularly if you are assigning a value to the property based on user input. If the user enters an invalid URL and there is no error handler, it could result in an exception which terminates the application.

Data Type

String

Example

```
' Setup error handling since the control will throw an error
' if an invalid URL is specified

On Error Resume Next: Err.Clear
HttpClient1.URL = Text1.Text

' Check the Err object to see if an error has occurred, and
' if so, let the user know that the URL is invalid

If Err.Number <> 0 Then
    MsgBox "The specified URL is invalid", vbExclamation
    Text1.SetFocus
    Exit Sub
End If

' Reset error handling and connect to the server using the
' default property values that were updated when the URL
' property was set (ie: HostName, RemotePort, Resource, etc.)

On Error GoTo 0
nError = HttpClient1.Connect()

If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' Get the resource and store the data in a string buffer
nError = HttpClient1.GetData(HttpClient1.Resource, strBuffer)
```

See Also

HostAddress Property, HostName Property, Password Property, RemotePort Property, Resource Property, UserName Property, Connect Method

UserAgent Property

Gets and sets the current user agent value which identifies the application.

Syntax

object.**UserAgent** [= *UserAgent*]

Remarks

The **UserAgent** property identifies the application that is issuing the request to the server. This is a string value that should be defined using the following format:

```
Application[/Version] [(Additional Information)]
```

For example, if the name of your application is "MyProgram" and the current version is 2.0, then you could specify a user agent string as follows:

```
MyProgram/2.0
```

The additional information included with the user agent string should be enclosed in parenthesis and can include the operating system, version and build numbers for additional components as well as any other information that you wish to include. Multiple items should be separated by semicolons. It is recommended that the user agent string not be greater than 128 characters in length.

Data Type

String

See Also

HeaderField Property, HeaderValue Property

UserName Property

Gets and sets the current user name.

Syntax

object.**UserName** [= *username*]

Remarks

The **UserName** property specifies the username used to authenticate the user on the current server. This property must be set to access resources that are restricted on the server. Note that it is required to set both the **UserName** and **Password** properties to enable client authentication.

If your application needs to use OAuth 2.0 for authentication, it is recommended you set the **BearerToken** property, which will automatically set the correct authentication type.

Data Type

String

See Also

AuthType Property, BearerToken Property, Password Property, Authenticate Method, Connect Method

Version Property

Return the current version of the object.

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

Hypertext Transfer Protocol Control Methods

Method	Description
AddField	Add the form field and its value to the current form
AddFile	Append the contents of the file to the current form
AsyncGetFile	Download a file from the server to the local system in the background
AsyncPutFile	Upload a file from the local system to the server in the background
Authenticate	Authenticate the client session with a username and password
Cancel	Cancels the current blocking network operation
ClearForm	Remove all defined fields from the current form
ClearHeaders	Resets the the current request and response headers to their default values
CloseFile	Close the file that was opened on the server
Command	Send a custom command to the server
Connect	Establish a connection with a server
CreateFile	Create a file on the server
CreateForm	Create a new form, replacing the current form
DeleteField	Delete the form field and its value from the current form
DeleteFile	Remove a file on the server
Disconnect	Terminate the connection with a server
GetCookie	Return information about the specified cookie
GetData	Transfer data from the server and store it in a local buffer
GetFile	Copy a file from the server to the local system
GetFileSize	Return the size of the specified file on the server
GetFileTime	Return the modification date and time for specified file on the server
GetFirstHeader	Return the first response header field name and value
GetHeader	Return the value of the specified header field
GetNextHeader	Return the next response header field name and value
GetText	Retrieve a text resource from the server and store it in a string buffer
Initialize	Initialize the control and validate the runtime license key
OpenFile	Open a file on the server for reading
PatchData	Submit patch data to the server and return the response in a string buffer
PostData	Transfer data from a local buffer and stores it in a file on the server
PostFile	Post the contents of the specified file to a script executed on the server
PostJson	Post JSON formatted data to a script executed on the server

PostXml	Post XML formatted data to a script executed on the server
PutData	Transfer data from a local buffer to the server
PutFile	Copy a file from the local system to the server
PutText	Submit the contents of a string buffer to the server
Read	Return data read from the server
Reset	Reset the internal state of the control
SetCookie	Send the specified cookie to the server when a resource is requested
SetHeader	Set the value of a request header field
SubmitForm	Submit the current form to the server for processing
TaskAbort	Abort the specified asynchronous task
TaskDone	Determine if an asynchronous task has completed
TaskResume	Resume execution of an asynchronous task
TaskSuspend	Suspend execution of an asynchronous task
TaskWait	Wait for an asynchronous task to complete
Uninitialize	Uninitialize the control and release any system resources that were allocated
Write	Write data to the server

AddField Method

Add the form field and its value to the current form.

Syntax

object.**AddField**(*FieldName*, *FieldData*, [*FieldLength*])

Parameters

FieldName

A string which specifies the name of the field to add to the form.

FieldData

A string or byte array which specifies the data for the form field.

FieldLength

An integer value which specifies the length of the field data in characters or bytes, depending on whether the field data was specified as a string or byte array. If this optional argument is omitted, the complete string or byte array 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 **AddField** method is used to add a field and its associated value to a form created using the **CreateForm** method. If the field name has already been added to the form, the previous value is deleted and replaced by the new value.

Example

```
HttpClient1.CreateForm "/login/php", httpMethodPost, httpFormEncoded
HttpClient1.AddField "UserName", strUserName
HttpClient1.AddField "Password", strPassword

nError = HttpClient1.SubmitForm(strResult)
If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

AddField Method, AddFile Method, CreateForm Method, DeleteField Method, SubmitForm Method

AddFile Method

Append the contents of the file to the current form.

Syntax

object.**AddFile**(*FieldName*, *FileName*)

Parameters

FieldName

A string which specifies the name of the field to add to the form.

FileName

A string which specifies the name of the file.

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 **AddFile** method is used to add the contents of a file to a form created using the **CreateForm** method. If the field name has already been added to the form, the previous value is deleted and replaced by the new value.

See Also

AddField Method, AddFile Method, CreateForm Method, DeleteField Method, SubmitForm Method

AsyncGetFile Method

Download a file from the server to the local system in the background.

Syntax

object.**AsyncGetFile**(*LocalFile*, *RemoteFile*, [*Options*], [*Offset*])

Parameters

LocalFile

A string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

RemoteFile

A string that specifies the file on the server that will be transferred to the local system. The file pathing and name conventions must be that of the server.

Options

An optional numeric bitmask which specifies one or more options. This argument may be any one of the following values:

Value	Description
httpTransferDefault	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the remote file. If the Options argument is omitted, this is the transfer mode which will be used.
httpTransferConvert	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
httpTransferCompress	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled by setting the **Compression** property to True. This option is ignored if the ***Offset*** parameter is non-zero.

Offset

An optional byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully.

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 **AsyncGetFile** method will download the contents of a remote file to a file on the local system. It is similar to the **GetFile** method, however it retrieves the file using a background worker thread and

does not block the current working thread. This enables the application to continue to perform other operations while the file is being downloaded from the server. This method requires that you explicitly establish a connection using the **Connect** method. All background tasks will duplicate the active connection and use it establish a secondary connection with the server to perform the file transfer. If you wish to perform multiple asynchronous file transfers from different servers, you must create an instance of the control for each server.

After this method is called, the **OnTaskBegin** event will be fired, indicating that the background task has begun the process of connecting to the server and performing the file transfer. As the file is downloaded, the control will periodically invoke the **OnTaskRun** event handler. When the transfer has completed, the **OnTaskEnd** event will be fired. It is not required that you implement handlers for these events.

To determine when a transfer has completed without implementing any event handlers, periodically call the **TaskDone** method. If you wish to block the current thread and wait for the transfer to complete, call the **TaskWait** method. To stop a background file transfer that is in progress, call the **TaskAbort** method. This will signal the background worker thread to cancel the transfer and terminate the session.

This method can be called multiple times to download more than one file in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. The application should not make any assumptions about the sequence in which background transfers are performed or the order in which they may complete.

Example

```
' Establish a connection to the server
nError = HttpClient1.Connect(strHostName, 80)

If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' Download a file in the background
nError = HttpClient1.AsyncGetFile(strLocalFile, strRemoteFile)

If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

TaskId Property, AsyncPutFile Method, TaskAbort Method, TaskDone Method, TaskWait Method, OnTaskBegin Event, OnTaskEnd Event, OnTaskRun Event

AsyncPutFile Method

Upload a file from the local system to the server in the background.

Syntax

object.**AsyncPutFile**(*LocalFile*, *RemoteFile*, [*Options*], [*Offset*])

Parameters

LocalFile

A string that specifies the file on the local system that will be transferred to the server. The file pathing and name conventions must be that of the local host.

RemoteFile

A string that specifies the file on the server that will be created, overwritten or appended to. The file pathing and name conventions must be that of the server.

Options

An optional numeric bitmask which specifies one or more options. This argument may be any one of the following values:

Value	Description
httpTransferDefault	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the remote file. If the Options argument is omitted, this is the transfer mode which will be used.

Offset

An optional byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully.

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 **AsyncPutFile** method will upload the contents of a file on the local system to the server. It is similar to the **PutFile** method, however it retrieves the file using a background worker thread and does not block the current working thread. This enables the application to continue to perform other operations while the file is being uploaded to the server. This method requires that you explicitly establish a connection using the **Connect** method. All background tasks will duplicate the active connection and use it establish a secondary connection with the server to perform the file transfer. If you wish to perform multiple asynchronous file transfers from different servers, you must create an instance of the control for each server.

After this method is called, the **OnTaskBegin** event will be fired, indicating that the background task has begun the process of connecting to the server and performing the file transfer. As the file is uploaded, the control will periodically invoke the **OnTaskRun** event handler. When the transfer has completed, the **OnTaskEnd** event will be fired. It is not required that you implement handlers for these events.

To determine when a transfer has completed without implementing any event handlers, periodically

call the **TaskDone** method. If you wish to block the current thread and wait for the transfer to complete, call the **TaskWait** method. To stop a background file transfer that is in progress, call the **TaskAbort** method. This will signal the background worker thread to cancel the transfer and terminate the session.

This method can be called multiple times to upload more than one file in the background; however, most servers limit the number of simultaneous connections that can originate from a single IP address. The application should not make any assumptions about the sequence in which background transfers are performed or the order in which they may complete.

Example

```
' Establish a connection to the server
nError = HttpClient1.Connect(strHostName, 80, strUserName, strPassword)

If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' Upload a file in the background
nError = HttpClient1.AsyncPutFile(strLocalFile, strRemoteFile)

If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

TaskId Property, AsyncGetFile Method, TaskAbort Method, TaskDone Method, TaskWait Method, OnTaskBegin Event, OnTaskEnd Event, OnTaskRun Event

Authenticate Method

Authenticate the client session with a username and password.

Syntax

object.**Authenticate**(*UserName*, *Password*, [*AuthType*])

Parameters

UserName

A string which specifies the username used to authenticate the client session.

Password

A string which specifies the password which will be used to authenticate the client session with the server. Not all server resources require the client to authenticate the session. If you are using OAuth 2.0 authentication, this parameter specifies the bearer token.

AuthType

An optional value which specifies the authentication method. If this parameter is omitted, it will use the method specified by the **AuthType** property. It may be one of the following values:

Value	Description
httpAuthNone	No client authentication should be performed.
httpAuthBasic	The **Basic** authentication scheme should be used. This option is supported by all servers that support at least version 1.0 of the protocol. The user credentials are not encrypted and Basic authentication should not be used over standard (non-secure) connections. Most web services which use Basic authentication require the connection to be secure.
httpAuthBearer	The **Bearer** authentication scheme should be used. This authentication method does not require a user name and the **BearerToken** property must specify the OAuth 2.0 bearer token issued by the service provider. If the access token has expired, the request will fail with an authorization error. This function will not automatically refresh an expired token.

Return Value

A value of zero is returned if the method was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

This method will set the Authorization request header for the client session using the credentials provided by the caller. It will always override any custom Authorization header value which may have been previously set using the **SetHeader** method.

If both the **UserName** and **Password** parameters specify empty strings, the current authentication type will always be set to **httpAuthNone** regardless of the value of the **AuthType** parameter. This effectively clears the current user credentials for the client session.

If you provide a user name and password to the **Connect** method, or you set the **UserName** property and either the **Password** or **BearerToken** property prior to calling the **Connect** method, authentication will be automatically attempted at the time the connection is made. This method is only required if you do not provide user credentials when the connection is established and wish to authenticate the client session at a later time.

See Also

AuthType Property, BearerToken Property, Password Property, UserName Property, Connect Method
SetHeader 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

Disconnect Method, Reset Method, OnCancel Event

ClearForm Method

Remove all defined fields from the current form.

Syntax

object.**ClearForm**

Parameters

None.

Return Value

None.

See Also

AddField Method, AddFile Method, CreateForm Method, DeleteField Method, SubmitForm Method

ClearHeaders Method

Resets the the current request and response headers to their default values.

Syntax

object.**ClearHeaders**

Parameters

None.

Return Value

None.

Remarks

The **ClearHeaders** method removes all custom header values for the current client session and clears the response headers for the previous request. This method is useful when using persistent connections and you want to ensure that any custom header values from the previous request will not be included with subsequent requests made during the same session. Because this method clears all of the request headers for the client session, this means it will also remove any cookies which have been set using the **SetCookie** method.

This method will clear any authentication credentials or tokens specified for the current client session. If the next request you make requires authentication, you must provide those credentials again using the **Authenticate** method.

It is not necessary to use this method if the **KeepAlive** property is False and you are not using persistent connections. The headers for the previous request are automatically cleared when a new connection is established.

See Also

KeepAlive Property, Authenticate Method, Connect Method, GetCookie Method, GetHeader Method, SetCookie Method, SetHeader Method

CloseFile Method

Close the file that was opened on the server.

Syntax

object.**CloseFile**

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 CloseFile method is used to close a file that was opened using the **OpenFile** method, or created using the **CreateFile** method. It should be called before the client disconnects from the server.

See Also

CreateFile Method, OpenFile Method

Command Method

Send a custom command to the server.

Syntax

object.**Command**(*Command*, *Resource*, [*Parameters*], [*Options*])

Parameters

Command

A string which specifies the command to send. There are a number of standard commands which may be used, and there are extended commands which depend on the type of server that the client is connected to. Consult the protocol standard and/or the technical reference documentation for the server to determine what commands may be issued by a client application. An example of some common HTTP commands are:

Command	Description
GET	Return the contents of the specified resource. This command is recognized by all servers.
HEAD	Return only header information for the specified resource. This command is recognized by servers that support at least version 1.0 of the protocol.
POST	Post data to the specified resource. This command is recognized by servers that support at least version 1.0 of the protocol.
PUT	Create or replace the specified resource on the server. This command is recognized by servers that support at least version 1.0 of the protocol. Not all servers support this command.
DELETE	Delete the specified resource from the server. This command is recognized by servers that support at least version 1.1 of the protocol. Not all servers support this command.

Resource

A string which specifies the resource that the command is to be performed upon. The resource may be a file, such as an HTML document or an image, or it may be a script used to process data submitted by the client. Note that this argument specifies the name of the resource only, not a complete URL.

Parameters

An optional string or byte array which specifies one or more parameters to be sent along with the command. The parameter data is encoded according to the encoding type specified by the **Encoding** property. If the resource does not require any parameters, this argument should be omitted. Note that it is possible to pass binary data by specifying an array of bytes rather than a string as the argument.

Options

A numeric value which specifies one or more options. Currently this argument is reserved and should either be omitted, or a value of zero should always be used.

Return Value

A value of zero is returned if the command was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure. To determine the result code returned by the server

in response to the command, read the value of the **ResultCode** property.

Remarks

The **Command** method sends a command to the server and processes the result code sent back in response to that command. This method can be used to send custom commands to a server to take advantage of features or capabilities that may not be supported internally by the control.

Not all servers support all of the listed commands, and some commands may require specific changes to the server configuration. In particular, the PUT and DELETE commands typically require that configuration changes be made by the site administrator. All servers will support the use of the GET command, and all servers that support at least version 1.0 of the protocol will support the POST command.

The ***Parameters*** argument is used to pass additional information to the server when a resource is requested. This is most commonly used to provide information to scripts, similar to how arguments are used when executing a program from the command line. Unless the POST command is being executed, the data in the buffer will automatically be encoded using the current encoding mechanism specified for the client. By default, the data is URL encoded, which means that any spaces and non-printable characters are converted to printable characters before submitted to the server. The type of encoding that is performed can be changed by setting the **Encoding** property. Although the default encoding is appropriate for most applications, those that submit XML formatted data may need to change the encoding type.

Only one request may be in progress at one time for each client session. Use the **CloseFile** method to terminate the request after all of the data has been read from the server.

See Also

Encoding Property, ResultCode Property, ResultString Property, OnCommand Event

Connect Method

Establish a connection with a server.

Syntax

object.**Connect**([*RemoteHost*], [*RemotePort*], [*UserName*], [*Password*], [*Timeout*], [*Options*])

Parameters

RemoteHost

A string which specifies the host name or IP address of the server. If this argument is not specified, it defaults to the value of the **HostAddress** property if it is defined. Otherwise, it defaults to the value of the **HostName** property.

RemotePort

A number which specifies the port to connect to on the server. If this argument is not specified, it defaults to the value of the **RemotePort** property. A value of zero indicates that the default port number for this service should be used to establish the connection.

UserName

An optional string which specifies a username used to authenticate the client session. This argument is only required if access to the resource requires authentication. If this argument is omitted, it defaults to the value of the **UserName** property. An empty string means that no authentication will be performed.

Password

An optional string which specifies the password used to authenticate the client session. This argument is only required if access to the resource requires authentication. If this argument is omitted, it defaults to the value of the **Password** property. An empty string means that no password will be provided.

Timeout

The number of seconds that the client will wait for a response before failing the operation. If this argument is not specified, the value of the **Timeout** property will be used as the default.

Options

An unsigned integer that specifies one or more options. If this parameter is omitted, it defaults to the value of the **Options** property. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
httpOptionNoCache	This instructs the server to not return a cached copy of the resource. When connected to an HTTP 1.0 or earlier server, this directive may be ignored.
httpOptionKeepAlive	This instructs the server to maintain a persistent connection between requests. This can improve performance because it eliminates the need to establish a separate

	connection for each resource that is requested. If the server does not support the keep-alive option, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers.
httpOptionRedirect	This option specifies the client should automatically handle resource redirection. If the server indicates that the requested resource has moved to a new location, the client will close the current connection and request the resource from the new location. Note that it is possible that the redirected resource will be located on a different server.
httpOptionProxy	This option specifies the client should use the default proxy configuration for the local system. If the system is configured to use a proxy server, then the connection will be automatically established through that proxy; otherwise, a direct connection to the server is established. The local proxy configuration can be changed in the system settings or control panel.
httpOptionErrorData	This option specifies the client should return the content of an error response from the server, rather than returning an error code. Note that this option will disable automatic resource redirection, and should not be used with **httpOptionRedirect**.
httpOptionNoUserAgent	This option specifies the client should not include a User-

	Agent header with any requests made during the session. The user agent is a string which is used to identify the client application to the server. An application can provide its own custom user agent value using the **SetHeader** method.	
httpOptionHttp2	This option specifies the client should attempt a HTTP/2 connection with the server. If a connection cannot be established using HTTP/2 the client will attempt to connect using an earlier version of the protocol. The value of the **ProtocolVersion** property will be ignored when this option is used.	
&H400	httpOptionTunnel	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
&H800	httpOptionTrustedSite	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 the TLS protocol.
&H1000	httpOptionSecure	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol. The client will default to using TLS 1.2 or later for secure connections.

&H8000	httpOptionSecureFallback	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.
&H40000	httpOptionPreferIPv6	This option specifies the client should attempt to resolve a domain name to an IPv6 address. If the domain name has both an IPv4 and IPv6 address assigned to it, the default is to use the IPv4 address for compatibility purposes. Enabling this option forces the client to always use the IPv6 address if one is available. If the domain name does not have an assigned IPv4 address, the IPv6 address will always be used regardless if this option is specified.
&H100000	httpOptionHiResTimer	This option specifies the elapsed time for data transfers should be returned in milliseconds instead of seconds. This will return more accurate transfer times for smaller amounts of data being uploaded or downloaded using fast network connections.

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

If the **httpOptionKeepAlive** option is specified and the server does not support persistent connections, the client will automatically reconnect when each resource is requested. Although it will not provide any performance benefits, this allows the option to be used with all servers. This option is automatically enabled when using HTTP/2.

Under some circumstances, a server may forcibly close or abort a connection if the request fails. For example, this can happen when a malformed request is submitted to the server or a server-side error occurs. When this happens, the application will need to disconnect from the server and establish a new connection, even if the keep-alive option has been specified.

If your application specifies the **httpOptionHttp2** option, a secure connection using TLS 1.2 or later

will always be used. The minimum required platform for HTTP/2 support is Windows 10 (Build 1903) or Windows Server 2019. Earlier versions of Windows do not support the features required for a secure HTTP/2 connection. If the server only accepts earlier versions of the protocol, the client will attempt to automatically downgrade the request to HTTP/1.1.

See Also

AutoRedirect Property, HostAddress Property, HostName Property, Options Property, ProxyHost Property, ProxyPort Property, ProxyType Property, RemotePort Property, URL Property, Disconnect Method, OnConnect Event

CreateFile Method

Create a file on the server.

Syntax

object.**CreateFile**(*FileName*, *FileLength*)

Parameters

FileName

A string which specifies the name of the file being created on the server. The client must have the appropriate access rights to create the file or an error will be returned.

FileLength

A number which specifies the size of the file in bytes. This value must be greater than zero.

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 **CreateFile** method uses the PUT command to create the file. The server must support this command and the user must have the appropriate permission to create the specified file. If this method is successful, the client should then use the **Write** method to send the contents of the file to the server. Once all of the data has been written, the **CloseFile** method should be called to close the file and complete the operation. Note that this method is typically only accepted by servers that support version 1.1 of the protocol or later.

When using **Write** to send the contents of the file to the server, it is recommended that the data be written in logical blocks that are no larger than 8,192 bytes in size. Attempting to write very large amounts of data in a single call can either cause the thread to block or, in the case of an asynchronous connection, return an error if the internal buffers cannot accommodate all of the data. To send the entire contents of a file at once, use the **PutData** method instead of calling **CreateFile**.

See Also

CloseFile Method, OpenFile Method, PutData Method, PutFile Method, Write Method

CreateForm Method

Create a new form, replacing the current form.

Syntax

object.**CreateForm**([*Action*], [*Method*], [*FormType*])

Parameters

Action

A string which specifies the name of the resource that the form data will be submitted to. Typically this is the name of a script that is executed on the server. If this argument is omitted, the value of the **FormAction** property is used as the default value. If the **FormAction** property is undefined, the value of the **Resource** property will be used as the default value.

Method

An integer value which specifies how the form data will be submitted to the server. This argument may be one of the following values:

Value	Description
httpMethodDefault	The form data should be submitted using the default method, using the GET command.
httpMethodGet	The form data should be submitted using the GET command. This method should be used when the amount of form data is relatively small. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used instead.
httpMethodPost	The form data should be submitted using the POST command. This is the preferred method of submitting larger amounts of form data. If the total amount of form data exceeds 2048 bytes, it is recommended that the POST method be used.

FormType

An integer value which specifies the type of form and how the data will be encoded when it is submitted to the server. This argument may be one of the following values:

Value	Description
httpFormDefault	The form data should be submitted using the default encoding method.
httpFormEncoded	The form data should be submitted as URL encoded values. This is typically used when the GET method is used to submit the data to the server.
httpFormMultipart	The form data should be submitted as multipart form data. This is typically used when the POST method is used to submit a file to the server. Note that the script must understand how to process multipart form data if this form type is specified.

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 **CreateForm** method is used to create a new form that will be populated with values and then submitted to the server for processing.

Example

```
HttpClient1.CreateForm "/login/php", httpMethodPost, httpFormEncoded
HttpClient1.AddField "UserName", strUserName
HttpClient1.AddField "Password", strPassword

nError = HttpClient1.SubmitForm(strResult)
If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

AddField Method, AddFile Method, CreateForm Method, DeleteField Method, SubmitForm Method

DeleteField Method

Delete the form field and its value from the current form.

Syntax

object.**DeleteField**(*FieldName*)

Parameters

FieldName

A string which specifies the name of the field to remove from the form.

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 **DeleteField** method is used to remove a field and its associated value from the current form.

See Also

AddField Method, AddFile Method, CreateForm Method, DeleteField Method, SubmitForm Method

DeleteFile Method

Remove a file on the server.

Syntax

object.**DeleteFile**(*FileName*)

Parameters

FileName

An string value which specifies the name of the resource or file to be deleted.

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 **DeleteFile** method deletes an existing file from the HTTP server using the DELETE command. This command is typically only accepted by servers that support version 1.1 of the protocol or later. Note that this method requires that the server be configured to permit file deletion and that the user has the appropriate permission to remove the file.

See Also

CreateFile Method, GetFile Method, OpenFile Method, PutFile Method

Disconnect Method

Terminate the connection with a server.

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

IsConnected Property, Connect Method, OnDisconnect Event

GetCookie Method

Return information about the specified cookie.

Syntax

object.**GetCookie**(*CookieName*, *CookieValue*, [*CookiePath*], [*CookieDomain*], [*CookieExpires*], [*CookieFlags*])

Parameters

CookieName

A string which specifies the name of the cookie to return information about. To obtain a list of cookies which have been set by the server, use the **CookieCount** and **CookieName** properties.

CookieValue

A string which will contain the value of the cookie when the method returns. This parameter must be passed by reference.

CookiePath

An optional string argument which will contain the cookie path when the method returns. This specifies a path for the resources where the cookie should be used. For example, a path of "/" indicates that the cookie should be provided for all resources requested from the server. A path of "/data" would mean that the cookie should be included if the resource is found in the /data folder or a sub-folder, such as /data/projections.asp. However, the cookie would not be provided if the resource /info/status.asp was requested, since it is not in the /data path. The cookie should only be sent to the server if the resource being requested is located in the directory or subdirectory of this path. This parameter must be passed by reference. If this information is not required, the argument can be omitted.

CookieDomain

An optional string argument which will contain the domain that the cookie is valid for. Matches are made by comparing the name of the server against the domain name specified in the cookie. If the domain is example.com, then any server in the example.com domain would match; for example, both shipping.example.com and orders.example.com would match the domain value. However, if the cookie domain was orders.example.com, then the cookie would only be sent if the resource was requested from orders.example.com, not if the resource was located on shipping.example.com or www.example.com. This parameter must be passed by reference. If this information is not required, the argument can be omitted.

CookieExpires

An optional date value which specifies when the cookie expires and should no longer be sent to the server when requesting a resource in the path specified by the ***CookiePath*** value. This is only valid for persistent cookies, since session cookies are automatically deleted when the client application terminates. The time is always expressed as Coordinated Universal Time. This parameter must be passed by reference. If this information is not required, the argument can be omitted.

CookieFlags

An optional integer value which provides status information about the cookie. A value of zero indicates that there are no special status flags for the cookie. This parameter must be passed by reference. This argument may be omitted if the information is not required. The following values are currently defined:

Value	Description

httpCookieSecure	This flag specifies that the cookie should only be provided to the server if the connection is secure.
httpCookieSession	This flag specifies that the cookie should only be used for the current application session and should not be stored permanently on the local system.

Return Value

This method returns a Boolean value. A value of true is returned if the cookie name is valid. Otherwise, a value of false is returned, which indicates that a cookie with that name does not exist.

Remarks

The Hypertext Transfer Protocol uses special tokens called "cookies" to maintain persistent state information between requests for a resource. These cookies are exchanged between the client and server by setting specific header fields. When a server wants the client to use a cookie, it will include a header field named Set-Cookie in the response header when the client requests a resource. The client can then take this cookie and store it, either temporarily in memory or permanently in a file on the local system. The next time that the client requests a resource from that server, it can send the cookie back to the server by setting the Cookie header field. The **GetCookie** method searches for a cookie set by the server in the Set-Cookie header field. The **SetCookie** method creates or modifies the Cookie header field for the next resource requested by the client.

There are two general types of cookies that are used by servers. Session cookies exist only for the duration of the client session; they are stored in memory and not saved in any kind of permanent storage. When the client application terminates, session cookies are deleted and no longer used. Persistent cookies are stored on the local system and are used by the client until their expiration time.

It is the responsibility of the client application to store persistent cookies and determine if a cookie meets the criteria required to be submitted to the server. If the application wishes to send the cookie, it can use the **SetCookie** method and specify the cookie name and value.

See Also

CookieCount Property, CookieName Property, CookieValue Property, ClearHeaders Method, SetCookie Method

GetData Method

Retrieve data from the server and store it in a local buffer.

Syntax

object.**GetData**(*Resource*, *Buffer*, [*Length*], [*Options*])

Parameters

Resource

A string that specifies the resource on the server that will be accessed. If the resource specifies a file, then the contents of the file will be returned to the server. If the resource specifies a script or other executable content, it will be executed and the output will be transferred to the local system. The file pathing and name conventions must be that of the server.

Buffer

This parameter specifies the local buffer that the data will be stored in. If the variable is a String type, then the data will be returned as a string of characters. This is the most appropriate data type to use if the file on the server is a text file. If the remote file contains binary data, it is recommended that a Byte array variable be specified as the argument to this method.

Length

An optional integer argument that will contain the number of bytes copied into the buffer when the method returns.

Options

An optional integer value which specifies one or more options. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description
httpTransferDefault	The default transfer mode. The resource data is downloaded to the local system exactly as it is stored on the server. If you are requesting a text-based resource, the data may use a different end-of-line character sequence. For example, the end-of-line character may be a single linefeed character instead of a carriage return and linefeed pair.
httpTransferConvert	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
httpTransferCompress	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled by setting the **Compression** property to True.
httpTransferErrorData	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is

	returned to the client and the method will succeed.

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 **GetData** method transfers data from the server to the local system, storing it in the specified buffer . This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

When encountering a server error during a request, the **GetData** method normally returns an error code, and no data is copied into the caller-provided buffer. The error code reflects the general cause of the failure, allowing the application to handle this error condition appropriately. Additionally, servers may provide further details about the failure, such as XML or JSON formatted data containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the **httpTransferErrorData** option. When this option is enabled, the behavior of **GetData** changes; it does not return an error code for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the buffer provided by the caller. If this option is used, your application should check the **ResultCode** property to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

Specifying the **httpTransferCompress** option does not guarantee that the data returned by the server will actually be compressed, it only informs the server that the client is willing to accept compressed data. Whether or not a particular resource is compressed depends on the server configuration, and the server may decide to only compress certain types of resources, such as text files.

If compression has been enabled and the server returns compressed data, it will be automatically expanded before being returned to the caller. If the application is using the **OnProgress** event to determine the amount of data being returned by the server, it is important to keep in mind that the values reflect the size of the compressed data.

See Also

Compression Property, GetFile Method, PutData Method, PutFile Method, OnProgress Event

GetFile Method

Copy a file from the server to the local system.

Syntax

object.**GetFile**(*LocalFile*, *RemoteFile*, [*Options*], [*Offset*])

Parameters

LocalFile

A string that specifies the file on the local system that will be created, overwritten or appended to. The file pathing and name conventions must be that of the local host.

RemoteFile

A string that specifies the file on the server that will be transferred to the local system. The file pathing and name conventions must be that of the server.

Options

An optional numeric value which specifies one or more options. This argument may be any one of the following values:

Value	Description
httpTransferDefault	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the remote file. If the Options argument is omitted, this is the transfer mode which will be used.
httpTransferConvert	If the resource being downloaded from the server is textual, the data is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences.
httpTransferCompress	This option informs the server that the client is willing to accept compressed data. If the server supports compression for the specified resource, then the data will be automatically expanded before being returned to the caller. This option is selected by default if compression has been enabled by setting the **Compression** property to True. This option is ignored if the ***Offset*** parameter is non-zero.
httpTransferErrorData	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method will succeed. If this option is used, your application should check the **ResultCode** property to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

Offset

An optional integer value that specifies a byte offset into the file. If this value is greater than zero, the server must support the ability to specify a byte range with the request to download the file, otherwise this method will fail.

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 **GetFile** method copies an existing file from the server to the local system. This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

Specifying the **httpTransferCompress** option does not guarantee that the data returned by the server will actually be compressed, it only informs the server that the client is willing to accept compressed data. Whether or not a particular resource is compressed depends on the server configuration, and the server may decide to only compress certain types of resources, such as text files.

If compression has been enabled and the server returns compressed data, it will be automatically expanded before being returned to the caller. If the application is using the **OnProgress** event to determine the amount of data being returned by the server, it is important to keep in mind that the values reflect the size of the compressed data.

See Also

Compression Property, GetData Method, PutData Method, PutFile Method, OnProgress Event

GetFileSize Method

Return the size of the specified file on the server.

Syntax

object.**GetFileSize**(*RemoteFile*, *FileSize*)

Parameters

RemoteFile

A string that specifies the name of the file on the server. The filename cannot contain any wildcard characters and must follow the naming conventions of the operating system the server is hosted on.

FileSize

A numeric variable which will be set to the size of the file on the server. Note that if the variable is not large enough to contain the file size, an overflow error will occur. This parameter must be passed by reference.

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 **GetFileSize** method uses the HEAD command to retrieve header information about the file without downloading the contents of the file itself. This requires that the server support at least version 1.0 of the protocol standard, or an error will be returned.

The server may not return a file size for some resources. This is typically the case with scripts that generate dynamic content because the server has no way of determining the size of the output generated by the script without actually executing it. The server may also not provide a file size for HTML documents which use server side includes (SSI) because that content is also dynamically created by the server. If the request to the server was successful and the file exists, but the server does not return a file size, the method will succeed but the file size returned to the caller will be zero.

When a request is made to the server for information about the file, the control will attempt to keep the connection alive, even if the **KeepAlive** property has not been set to True. This allows an application to request the file size and then download the file without having to write additional code to re-establish the connection. However, it is possible that the attempt to keep the connection open will fail. In that case, an error will be returned and the session will no longer be valid. If this happens, the method may still return a valid file size. To determine if an error occurred, check the value of the **LastError** property.

Note that if the file on the server is a text file, it is possible that the value returned by this method will not match the size of the file when it is downloaded to the local system. This is because different operating systems use different sequences of characters to mark the end of a line of text, and when a file is transferred in text mode, the end of line character sequence is automatically converted to a carriage return-linefeed, which is the convention used by the Windows platform.

Example

The following example demonstrates how to retrieve the size a file on the server:

```
Dim nFileSize As Long

nError = HttpClient1.GetFileSize(strFileName, nFileSize)
```

```
If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

MsgBox "The size of " & strFileName & " is " & nFileSize " bytes"
```

See Also

GetFileTime Method

GetFileTime Method

Return the modification date and time for specified file on the server.

Syntax

object.**GetFileTime**(*RemoteFile*, *FileDate*)

Parameters

RemoteFile

A string that specifies the name of the file on the server. The filename cannot contain any wildcard characters and must follow the naming conventions of the operating system the server is hosted on.

FileDate

A variable that will be set to the date and time that the file was last modified. The variable's data type may either be **Variant**, **String** or **Date**. This parameter must be passed by reference.

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 **GetFileTime** method uses the HEAD command to retrieve header information about the file without downloading the contents of the file itself. This requires that the server support at least version 1.0 of the protocol standard, or an error will be returned.

The server may not return a modification time for some resources. If the request to the server was successful and the file exists, but the server does not return a modification time, the method will return an empty string.

When a request is made to the server for information about the file, the control will attempt to keep the connection alive, even if the **KeepAlive** property has not been set to True. This allows an application to request the modification time and then download the file without having to write additional code to re-establish the connection. However, it is possible that the attempt to keep the connection open will fail. In that case, an error will be returned and the session will no longer be valid. If this happens, the method may still return a valid date and time. To determine if an error occurred, check the value of the **LastError** property.

The **Localize** property will determine if the returned file time is adjusted for the local timezone.

Example

The following example demonstrates how to retrieve the size a file on the server:

```
Dim dateFileTime As Date

nError = HttpClient1.GetFileTime(strFileName, dateFileTime)
If nError > 0 Then
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

MsgBox strFileName & " was modified on " & dateFileTime
```

See Also

Localize Property, GetFileSize Method

GetFirstHeader Method

Return the first response header field name and value.

Syntax

object.**GetFirstHeader**(*HeaderField*, *HeaderValue*)

Parameters

HeaderField

A string that will contain the name of the first header field returned by the server.

HeaderValue

A string that will contain the value of the specified header field when the method returns. This parameter must be passed by reference.

Return Value

This method returns a Boolean value. If the method succeeds, it will return True. If the method fails, it will return False.

Remarks

The **GetFirstHeader** method is used get the first header field name and value from the response header returned by the server. This method should only be called after the client has requested the resource. This method is typically used in conjunction with the **GetNextHeader** method to enumerate all of the response header fields returned by the server.

See Also

HeaderField Property, HeaderValue Property, GetNextHeader Method, SetHeader Method

GetHeader Method

Return the value of the specified header field

Syntax

object.**GetHeader**(*HeaderField*, *HeaderValue*)

Parameters

HeaderField

A string that specifies the name of the header field to obtain the value for.

HeaderValue

A string that will contain the value of the specified header field with the method returns. This parameter must be passed by reference.

Return Value

This method returns a Boolean value. If the method succeeds, it will return True. If the header field has not been defined, it will return False.

Remarks

The **GetHeader** method is used get the value of a specified header field from the response header returned by the server. This method should only be called after the client has requested the resource. To enumerate all of the header fields returned by the server, use the **GetFirstHeader** and **GetNextHeader** methods.

See Also

HeaderField Property, HeaderValue Property, ClearHeaders Method, GetNextHeader Method, SetHeader Method

GetNextHeader Method

Return the next response header field name and value.

Syntax

object.**GetNextHeader**(*HeaderField*, *HeaderValue*)

Parameters

HeaderField

A string that will contain the name of the first header field returned by the server.

HeaderValue

A string that will contain the value of the specified header field.

Return Value

This method returns a Boolean value. If the method succeeds, it will return True. If the method fails, it will return False.

Remarks

The **GetNextHeader** method is used get the next header field name and value from the response header returned by the server. This method should only be called after the client has requested the resource. This method is used in conjunction with the **GetFirstHeader** method to enumerate all of the response header fields returned by the server.

See Also

HeaderField Property, HeaderValue Property, GetFirstHeader Method, SetHeader Method

GetText Method

Retrieve a text resource from the server and store it in a string buffer.

Syntax

object.**GetText**(*Resource*, *Buffer*)

Parameters

Resource

A string that specifies the resource on the server that will be accessed. If the resource specifies a text file, then the contents of the file will be returned to the server. If the resource specifies a script or other executable content, it will be executed and the output will be stored in the provided buffer.

Buffer

This parameter is passed by reference and specifies the string buffer which will contain the text returned by the server. This parameter must be a String or Variant type which will reference a string when the method returns. This method will not accept a byte array as an argument.

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 **GetText** method is used to download a text file or retrieve the text output from a script and store the contents in a string buffer. Because binary data can include embedded null characters which would truncate the string, this method should only be used with text files or script output which is known to be textual. For example, it is safe to use this method when the server returns HTML or XML data, but should not be used if it returns an image or executable file. Always use the **GetData** method if you wish to retrieve binary data and store it in a byte array.

The text returned by the server is automatically converted to Unicode using the code page specified by the **CodePage** property. Text returned by a web server will typically use UTF-8 encoding, however some servers may return text content using their own locale. If you specify an incorrect code page, this can result in a conversion error.

This method will always attempt to normalize the end-of-line character sequence to use a carriage-return and linefeed (CRLF) pair. This can potentially result in a discrepancy between the size of a text file on the server and the actual length of the string buffer.

This method will attempt to indicate to the server that only textual data is acceptable, however some servers may ignore this constraint. If the server does return binary data in response to the request, the string buffer may be empty or contain unprintable characters as the result of attempting to convert the data to Unicode.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

CodePage Property, GetData Method, GetFile Method, PostData Method, PutData Method, PutFile Method, OnProgress Event

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 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 httpClient = CreateObject("SocketTools.HttpClient.11")

nError = httpClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize Method

OpenFile Method

Open a file on the server for reading.

Syntax

object.**OpenFile**(*FileName*)

Parameters

FileName

A string which specifies the name of the file being opened on the server. The client must have the appropriate access rights to open the file for reading or an error will be returned. It may be required that the **UserName** and **Password** properties be set to authenticate the client session so that access to the resource is permitted.

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 **OpenFile** method uses the GET command to access the file. If this method is successful, the client should then use the **Read** method to read the contents of the file from the server. Once all of the data has been read, the **CloseFile** method should be called to close the file and complete the operation. If the file being opened is not an HTML or text document, then it's recommended that you read the data into a byte array.

This method should not be used to post data to a script or other executable resource on the server. If you wish to post data to a script, then the **PostData** method should be used instead.

See Also

CloseFile Method, CreateFile Method, GetData Method, GetFile Method, PostData Method, Read Method

PatchData Method

Submits patch data to the server and returns the result in a string buffer provided by the caller.

Syntax

object.**PatchData**(*Resource*, *PatchData*, *Buffer*, [*Options*])

Parameters

Resource

A string that specifies the resource that the patch data will be submitted to. Typically this is the name of an executable script.

PatchData

A string that contains the patch information. Typically this is XML or JSON formatted data which contains the information that should be used to update the specified resource.

Buffer

A string or byte array that will contain the output generated by the script. Typically this is XML or JSON content which is generated by the script as a result of processing the data that was posted to it.

Options

An optional integer value which specifies one or more options. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description
httpPatchDefault	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
httpPatchConvert	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.
httpPatchErrorData	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method will succeed.

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 **PatchData** method is used to submit XML or JSON formatted patch data to a service, and then returns a copy of the response from the server into a local string buffer. This method will not perform any encoding and will not automatically define the type of patch data being submitted. Your application is responsible for specifying the content type for the patch data, and ensuring that the

XML or JSON data that is being submitted to the server is formatted correctly.

This method sends a PATCH command to the server, which is similar to a POST or PUT request. It is used to make partial updates to a resource, rather than creating or replacing it entirely. The format of the patch data is specific to the service being used. If the resource being patched does not exist, the behavior is defined by the server. If enough information is provided, it may choose to create the resource just as if a PUT command was used, or it may return an error.

Your application should use the **SetHeader** method to define the Content-Type header prior to calling the **PatchData** method. One of the most common formats used is the JSON Merge Patch which is defined in RFC 7396. The value for the Content-Type header for this patch format is "application/merge-patch+json". Refer to your service API documentation to determine what patch formats are acceptable, along with any additional header values that must be defined.

When encountering a server error during a request, the **PatchData** method normally returns an error code, and no data is copied into the caller-provided buffer. The error code reflects the general cause of the failure, allowing the application to handle this error condition appropriately. If the PATCH request fails, servers may also provide further details about the failure, such as XML or JSON formatted data containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the **httpPatchErrorData** option. When this option is enabled, the behavior of **PatchData** changes; it does not return an error code for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should check the value of the **ResultCode** property to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This method will cause the current thread to block until the operation completes, a timeout occurs or the post is canceled. During the operation, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

GetData Method, PostData Method, PutFile Method, OnProgress Event

PostData Method

Submits the contents of the specified buffer to a script on the server.

Syntax

object.**PostData**(*Resource*, *ResourceData*, *Buffer*, [*Options*])

Parameters

Resource

A string that specifies the resource that the data will be posted to on the server. Typically this is the name of an executable script.

ResourceData

A string or byte array that contains the data which will be provided to the script. If the script expects binary data, it is recommended that a byte array be used.

Buffer

A string or byte array that will contain the output generated by the script. Typically this is HTML content which is generated by the script as a result of processing the data that was posted to it.

Options

An optional integer value which specifies one or more options. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description
httpPostDefault	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
httpPostConvert	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.
httpPostMultipart	The contents of the buffer being sent to the server consists of multipart form data. This causes the Content-Type request header field to be set to multipart/form-data and the contents of the buffer will be sent as-is without any encoding.
httpPostErrorData	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method will succeed.

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 **PostData** method uses the POST command to submit the contents of the specified buffer to a

script on the server and returns the result in a string or byte array provided by the caller. This method will cause the current thread to block until the operation completes, a timeout occurs or the post is canceled. During the operation, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

When encountering a server error during a request, the **PostData** method normally returns an error code, and no data is copied into the caller-provided buffer. The error code reflects the general cause of the failure, allowing the application to handle this error condition appropriately. If the POST request fails, servers may also provide further details about the failure, such as XML or JSON formatted data containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the **httpPostErrorData** option. When this option is enabled, the behavior of **PostData** changes; it does not return an error code for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should check the value of the **ResultCode** property to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

If you need to submit XML formatted data to the server for processing, it is recommended that you use the **PostXml** method. There is also a **PostJson** method for submitting JSON formatted data. Both methods ensure that the data is sent to the server using the correct content type and encoding.

See Also

GetData Method, GetFile Method, PatchData Method, PostFile Method, PostJson Method, PostXml Method, PutData Method, PutFile Method, OnProgress Event

PostFile Method

Post the contents of the specified file to a script executed on the server.

Syntax

object.**PostFile**(*LocalFile*, *Resource*, [*FieldName*], [*Options*])

Parameters

LocalFile

A string that specifies the file on the local system that will be transferred to the server. The file pathing and name conventions must be that of the local host.

Resource

A string that specifies the resource that the data will be posted to on the server. Typically this is the name of an executable script.

FieldName

An optional string argument that specifies the form field name that the script expects. If this argument is omitted or is an empty string, a default field name of "File1" is used.

Options

A numeric value which specifies one or more options. This argument may be any one of the following values:

Value	Description
httpTransferDefault	This option specifies the default transfer mode should be used. If the remote file exists, it will be overwritten with the contents of the uploaded file.

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 **PostFile** method posts the contents of a file to a script that is executed on the server. This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

This method is similar to the **PutFile** method in that it can be used to upload the contents of a local file to a server. However, instead of using the PUT command, the POST command is used to send the file data to a script that is executed on the server. This method has the advantage of not requiring any special configuration settings on the server, however it does require that the script be able to process multipart/form-data as defined in RFC 2388.

To support uploading files from a form on a webpage, the FILE input type is used along with the action that specifies the script that will accept the file data and process it. For example, the HTML code could look like this:

```
<form action="/cgi-bin/upload.cgi" method="post" enctype="multipart/form-
data">
<input type="file" name="datafile" size="20">
<input type="submit">
```

```
</form>
```

In this example, the script /cgi-bin/upload.cgi is responsible for processing the file data that is posted by the client, and the form field name "datafile" is used. The user can select a file, and when the Submit button is clicked, the file data is posted to the script. To simulate this using the **PostFile** method, the ***LocalFile*** argument should be set to the name of the local file that will be posted to the server. The ***Resource*** argument should be the name of the script, in this case "/cgi-bin/upload.cgi". The ***FieldName*** argument should be specified as the string "datafile" to match the name of the field used by the form.

Note that the **PostFile** function always submits the file contents as multipart/form-data with the content type set to application/octet-stream. The script that accepts the posted data must be able to parse the multipart header block and correctly process 8-bit data. If the script assumes that the data will be posted using a specific encoding type such as base64, then the file data may not be accepted or may be corrupted by the script.

See Also

GetData Method, GetFile Method, PostData Method, PutData Method, PutFile Method, OnProgress Event

PostJson Method

Submits JSON formatted data to the server and returns the result in a buffer provided by the caller.

Syntax

object.**PostJson**(*Resource*, *JsonData*, *Buffer*, [*Options*])

Parameters

Resource

A string that specifies the resource that the data will be posted to on the server. Typically this is the name of an executable script.

XmlData

A string that contains the JSON formatted data which will be provided to the script.

Buffer

A string or byte array that will contain the output generated by the script. Typically this is HTML content which is generated by the script as a result of processing the data that was posted to it.

Options

An optional integer value which specifies one or more options. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description
httpPostDefault	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
httpPostConvert	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.
httpPostErrorData	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method will succeed.

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 **PostJson** method is used to submit JSON formatted data to a script that executes on the server and then copy the output from that script into a local buffer. This function automatically sets the correct content type and encoding required for submitting JSON data to a server, however it does not parse the JSON data itself to ensure that it is well-formed. Your application is responsible for ensuring that the JSON data that is being submitted to the server is formatted correctly.

When encountering a server error during a request, the **PostJson** method normally returns an error

code, and no data is copied into the caller-provided buffer. The error code reflects the general cause of the failure, allowing the application to handle this error condition appropriately. If the POST request fails, servers may also provide further details about the failure, such as XML or JSON formatted data containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the **httpPostErrorData** option. When this option is enabled, the behavior of **PostJson** changes; it does not return an error code for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should check the value of the **ResultCode** property to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This method will cause the current thread to block until the operation completes, a timeout occurs or the post is canceled. During the operation, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

GetData Method, GetFile Method, PostData Method, PostFile Method, PostXml Method, PutData Method, PutFile Method, OnProgress Event

PostXml Method

Submits XML formatted data to the server and returns the result in a buffer provided by the caller.

Syntax

object.**PostXml**(*Resource*, *XmlData*, *Buffer*, [*Options*])

Parameters

Resource

A string that specifies the resource that the data will be posted to on the server. Typically this is the name of an executable script.

XmlData

A string that contains the XML formatted data which will be provided to the script.

Buffer

A string or byte array that will contain the output generated by the script. Typically this is HTML content which is generated by the script as a result of processing the data that was posted to it.

Options

An optional integer value which specifies one or more options. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description
httpPostDefault	The default post mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
httpPostConvert	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.
4	httpPostErrorData	This option causes the client to accept error data from the server if the request fails. If this option is specified, an error response from the server will not cause the method to fail. Instead, the response is returned to the client and the method

		will succeed.

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 **PostXml** method is used to submit XML formatted data to a script that executes on the server and then copy the output from that script into a local buffer. This function automatically sets the correct content type and encoding required for submitting XML data to a server, however it does not parse the XML data itself to ensure that it is well-formed. Your application is responsible for ensuring that the XML data that is being submitted to the server is formatted correctly.

When encountering a server error during a request, the **PostXml** method normally returns an error code, and no data is copied into the caller-provided buffer. The error code reflects the general cause of the failure, allowing the application to handle this error condition appropriately. If the POST request fails, servers may also provide further details about the failure, such as XML or JSON formatted data containing specific error codes or diagnostic messages.

To capture this error information, you can utilize the **httpPostErrorData** option. When this option is enabled, the behavior of **PostXml** changes; it does not return an error code for server error statuses. Instead, any error data provided in the server's response, regardless of its format, is copied into the result buffer provided by the caller. If this option is used, your application should check the value of the **ResultCode** property to obtain the HTTP status code returned by the server. This will enable you to determine if the operation was successful.

This method will cause the current thread to block until the operation completes, a timeout occurs or the post is canceled. During the operation, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

GetData Method, GetFile Method, PostData Method, PostFile Method, PostJson Method, PutData Method, PutFile Method, OnProgress Event

PutData Method

Transfer data from a local buffer to the server.

Syntax

object.**PutData**(*Resource*, *Buffer*, [*Length*], [*Reserved*])

Parameters

Resource

A string that specifies the resource on the server that will receive the data being transferred. If the resource is a file on the server, the contents will be replaced with the data provided in the buffer.

Buffer

This parameter specifies the local buffer that the data will be copied from. 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 file on the server is a text file. If the remote file should contain binary data, it is recommended that a **Byte** array variable be specified as the argument to this method.

Length

An optional integer argument that specifies the amount of data to be copied from the buffer. If this argument is omitted, the entire contents of the buffer is transferred to the server.

Reserved

An argument reserved for future expansion. This argument should always be omitted or specified as a numeric value of zero.

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 **PutData** method transfers data from a local buffer and submits to the server using the PUT command. If a String variable is used for the buffer, the text will be automatically UTF-8 encoded before it is submitted to the server. If you need to submit binary data, you should always use a **Byte** array as the buffer rather than a **String**. If you need to use something other than UTF-8 encoding, use the **PutText** method in combination with the **CodePage** parameter to specify an alternate encoding.

Not all servers will accept data submitted using this method, and some may require that specific configuration changes be made to the server in order to support the PUT command. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself by setting the **UserName** and **Password** properties prior to uploading the data.

If the ***Buffer*** parameter is a **String** type, this method presumes that it only contains text and will automatically convert the contents to UTF-8 encoded text. If the string contains binary data, this encoding can corrupt the data. To prevent this conversion, convert the string to a byte array using the **StrConv** function.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

GetData Method, GetFile Method, PostData Method, PutFile Method, OnProgress Event

PutFile Method

Copy a file from the local system to the server.

Syntax

object.**PutFile**(*LocalFile*, *RemoteFile*, [*Options*], [*Offset*])

Parameters

LocalFile

A string that specifies the file on the local system that will be transferred from the local system. The file pathing and name conventions must be that of the local host.

RemoteFile

A string that specifies the file on the server that will be created or overwritten. The file pathing and name conventions must be that of the server.

Options

An optional numeric value which specifies one or more options. This argument may be any one of the following values:

Value	Description
httpTransferDefault	This option specifies the default transfer mode should be used. If the local file exists, it will be overwritten with the contents of the remote file. If the Options argument is omitted, this is the transfer mode which will be used.

Offset

An optional byte offset which specifies where the file transfer should begin. The default value of zero specifies that the file transfer should start at the beginning of the file. A value greater than zero is typically used to restart a transfer that has not completed successfully.

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 **PutFile** method is used to transfer a file from the local system to a server using the PUT command. Not all servers permit files to be uploaded using this method, and some may require that specific configuration changes be made to the server in order to support this functionality. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself by setting the **UserName** and **Password** properties prior to uploading the file.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

GetData Method, GetFile Method, PostFile Method, PutData Method, OnProgress Event

PutText Method

Submit the contents of a string buffer to the server.

Syntax

object.**PutText**(*RemoteFile*, *Buffer*)

Parameters

RemoteFile

A string that specifies the name of a file on the server that will be downloaded. The file pathing and name conventions must be that of the server.

Buffer

A string which contains the text to be stored on the server. This method will not accept a Byte array as an argument.

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 **PutText** method is used to submit the contents of a string to the server using the PUT command. Although a String variable may contain binary data, this method should only be used with strings which contain printable text. Always use the **PutData** method if you wish to submit binary data, using a Byte array instead of a String variable.

Not all servers will accept data submitted using this method, and some may require that specific configuration changes be made to the server in order to support the PUT command. Consult your server's technical reference documentation to see if it supports the PUT command, and if so, what must be done to enable it. It may be required that the client authenticate itself by setting the **UserName** and **Password** properties prior to uploading the data

The text submitted to the server is automatically converted from Unicode using the code page specified by the **CodePage** property. By default, text will be automatically converted to use UTF-8 encoding, however you can change this if you prefer to send the data using a different localized encoding. In most cases it is recommended you use UTF-8 to ensure the broadest compatibility with other applications.

This method will cause the current thread to block until the file transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

CodePage Property, GetData Method, GetText Method, PutData Method, OnProgress Event

Read Method

Return data read from the server.

Syntax

object.**Read**(*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, 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 server 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 server, 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 UTF-8 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

IsConnected Property, IsReadable Property, Write Method, OnRead Event, OnWrite 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.

The **Reset** and **Uninitialize** methods will abort all active background transfers and wait for those tasks to complete before returning to the caller. It is recommended that your application explicitly wait for background transfers to complete or abort them using this method before allowing the program to terminate. This will ensure that your program can perform any necessary cleanup operations. If there are active background tasks running at the time that the control instance is destroyed, it can force the control to stop those worker threads immediately without waiting for them to terminate gracefully.

See Also

Cancel Method, Initialize Method, Uninitialize Method

SetCookie Method

Send the specified cookie to the server when a resource is requested.

Syntax

object.**SetCookie**(*CookieName*, *CookieValue*)

Parameters

CookieName

A string which specifies the name of the cookie that is to be sent to the server when the next resource is requested.

CookieValue

A string which specifies the value of the cookie. To delete a cookie that has been previously set, this parameter should be an empty string.

Return Value

This method returns a Boolean value. A value of true is returned if the cookie has been set. Otherwise, a value of false is returned, which indicates that the cookie could not be created.

Remarks

The **SetCookie** method submits the cookie name and value to the server when a resource is requested or data is posted to a script. For more information about cookies and how they are used, refer to the **GetCookie** method.

See Also

CookieCount Property, CookieName Property, CookieValue Property, ClearHeaders Method, GetCookie Method

SetHeader Method

Set the value of a request header field.

Syntax

object.**SetHeader** *HeaderField*, *HeaderValue*

Parameters

HeaderField

A string that specifies the name of the request header field.

HeaderValue

A string that specifies the value associated with the given header field. If this argument is set to an empty string, the field is deleted from the request header.

Return Value

This method returns a Boolean value. If the method succeeds, it will return True. If the header field cannot be created or modified, it will return False.

Remarks

The **SetHeader** method is used to set the values of specific fields in the HTTP request header. This method should be called before the client has requested the resource. Some headers are automatically generated by methods that send resource requests. Some of these are supplied by the requesting methods only if the application has not previously defined the header. For others, the requesting method overrides what the application may have defined. The affected headers include:

- The **Accept** header value is generated with a value of ***/*** for most requests unless it has already been assigned. This tells the server that the client will accept all content types. Changing this header may cause requests to fail if the resource does not return a data type that matches the specified value.
- The **Authorization** header value is generated automatically if authentication has been specified. Applications should not modify this header value directly, set the **UserName** and **Password** properties instead.
- The **Connection** header value is generated in accordance with the settings of the **KeepAlive** property and the version of the HTTP protocol being used. This header is not used with HTTP 1.0 connections that do not support persistent client sessions.
- The **Content-Length** header value is generated automatically for POST and PUT requests. Applications should not modify this header value directly.
- The **Content-Type** header value is generated with a value of **application/x-www-form-urlencoded** for POST requests unless the **EncodingProperty** has been set to prevent URL-encoding of data. If a file is being uploaded using the PUT method, the header value is set according to the type of data that is being sent to the server. Applications should not modify this header value directly.
- The **Host** header value is assigned the host name of the server that was specified when the connection was established. It is not recommended that applications change this header value because it may yield unexpected results on servers that use virtual hosting.
- The **Proxy-Authorization** header value is generated for proxy connections which a username

and password has been specified. Applications should not modify this header value directly, set the **ProxyUser** and **ProxyPassword** properties instead.

See Also

HeaderField Property, HeaderValue Property, ClearHeaders Method GetFirstHeader Method, GetHeader Method, GetNextHeader Method,

SubmitForm Method

Submits the current form to the server for processing.

Syntax

object.**SubmitForm**(*Buffer*, [*Length*], [*Options*])

Parameters

Buffer

A string or byte array that will contain the output generated by the script. Typically this is HTML content which is generated by the script as a result of processing the data that was posted to it.

Length

An optional integer variable which specifies the maximum number of characters or bytes to be copied into the buffer. This variable will be updated with the actual number of characters or bytes copied when the method returns.

Options

An optional integer value which specifies one or more options. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description
httpSubmitDefault	The default submission mode. The contents of the buffer are encoded and sent as standard form data. The data returned by the server is copied to the result buffer exactly as it is returned from the server.
httpSubmitConvert	If the data being returned from the server is textual, it is automatically converted so that the end of line character sequence is compatible with the Windows platform. Individual carriage return or linefeed characters are converted to carriage return/linefeed character sequences. Note that this option does not have any effect on the form data being submitted to the server, only on the data returned by the server.

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 **SubmitForm** method submits the current form data to a script on the server and returns the result in a string or byte array provided by the caller. This method will cause the current thread to block until the operation completes, a timeout occurs or the post is canceled. During the operation, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

Example

```
HttpClient1.CreateForm "/login/php", httpMethodPost, httpFormEncoded
HttpClient1.AddField "UserName", strUserName
HttpClient1.AddField "Password", strPassword

nError = HttpClient1.SubmitForm(strResult)
If nError > 0 Then
```

```
    MsgBox HttpClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

AddField Method, AddFile Method, CreateForm Method, DeleteField Method, SubmitForm Method

TaskAbort Method

Abort the specified asynchronous task.

Syntax

object.**TaskAbort** ([*TaskId*], [*Milliseconds*])

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

Milliseconds

An optional integer value that specifies the number of milliseconds to wait for the background task to abort.

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 **TaskAbort** method signals the background worker thread associated with the task ID to abort the current operation and terminate as soon as possible. If the ***TaskId*** parameter is omitted, this method will abort all active background file transfers, otherwise it will only abort the specified task. If the ***Milliseconds*** parameter is omitted or has a value of zero, the method returns immediately after the background thread has been signaled. If the ***Milliseconds*** parameter is non-zero, the method will wait that amount of time for the background thread to terminate.

The **Reset** and **Uninitialize** methods will abort all active background transfers and wait for those tasks to complete before returning to the caller. It is recommended that your application explicitly wait for background transfers to complete or abort them using this method before allowing the program to terminate. This will ensure that your program can perform any necessary cleanup operations. If there are active background tasks running at the time that the control instance is destroyed, it can force the control to stop those worker threads immediately without waiting for them to terminate gracefully.

See Also

TaskCount Property, TaskList Property, TaskDone Method, TaskWait Method

TaskDone Method

Determine if an asynchronous task has completed.

Syntax

object.**TaskDone** ([*TaskId*])

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

Return Value

A Boolean value that specifies if the task has completed. A return value of **True** specifies that the background task has completed. A return value of **False** specifies that the background task is active.

Remarks

The **TaskDone** method is used to determine if the specified asynchronous task has completed. If the *TaskId* parameter is omitted, the method will check the status of the last background task that was started.

If you use this method to poll the status of a background task from within the main UI thread, you must ensure that Windows messages are processed so that the application remains responsive to the end-user. To check if a background transfer has completed, it is recommended that you use a timer to periodically call this method rather than calling it repeatedly within a loop.

To determine if the task completed successfully, the **TaskWait** method will provide the last error code associated with the task. Note that if this method returns **True**, it is guaranteed that calling **TaskWait** using the same task ID will return the error code to the caller immediately without causing the application to block.

See Also

TaskCount Property, TaskId Property, TaskList Property, TaskAbort Method, TaskWait Method

TaskResume Method

Resume execution of an asynchronous task.

Syntax

object.**TaskResume** (*TaskId*)

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

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 **TaskResume** method resumes execution of the background worker thread that was previously suspended using the **TaskSuspend** method. If the ***TaskId*** parameter is omitted, the method will resume execution of the last background task that was started.

See Also

TaskId Property, TaskSuspend Method, TaskWait Method

TaskSuspend Method

Suspend execution of an asynchronous task.

Syntax

object.**TaskSuspend** (*TaskId*)

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

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 **TaskSuspend** method will suspend execution of the background worker thread associated with the task. If the ***TaskId*** parameter is omitted, the method will suspend the last background task that was started.

Once the task has been suspended, it will no longer be scheduled for execution, however the client session will remain active and the task may be resumed using the **TaskResume** method. Note that if a task is suspended for a long period of time, the background operation may fail because it has exceeded the timeout period imposed by the server.

See Also

TaskId Property, TaskResume Method, TaskWait Method

TaskWait Method

Wait for an asynchronous task to complete.

Syntax

object.**TaskWait** ([*TaskId*], [*Milliseconds*], [*TimeElapsed*], [*TaskError*])

Parameters

TaskId

An optional integer value that specifies the unique identifier associated with a background task.

Milliseconds

An optional integer value that specifies the number of milliseconds to wait for the background task to complete.

TimeElapsed

An optional integer value passed by reference that will contain the elapsed time for the task in milliseconds when the method returns. If this information is not required, this parameter may be omitted. This parameter is ignored if the ***TaskId*** parameter is omitted.

TaskError

An optional integer value passed by reference that will contain the last error code for the task when the method returns. If this information is not required, this parameter may be omitted. This parameter is ignored if the ***TaskId*** parameter is omitted.

Return Value

A Boolean value that specifies if the task has completed. A return value of **True** specifies that the background task has completed. A return value of **False** specifies that the background task is active.

Remarks

The **TaskWait** method waits for the specified task to complete. If the ***TaskId*** parameter is omitted, this method will wait for all active tasks to complete. If a task ID is specified and the ***Milliseconds*** parameter is non-zero, this method will cause the current working thread to block until the task completes or the amount of time exceeds the number of milliseconds specified by the caller. If the ***Milliseconds*** parameter is zero, then this function will poll the status of the task and return immediately to the caller. If the ***Milliseconds*** parameter is omitted, then the method will wait an infinite period of time for the task to complete.

If the specified task has already completed at the time this method is called, the method will return immediately without causing the current thread to block. If the ***TimeElapsed*** parameter has been specified, it will contain the number of milliseconds that it took for the task to complete. If the ***TaskError*** parameter has been specified, it will contain the last error code value that was set by the worker thread before it terminated. If the ***TaskError*** value is zero, that means that the background task was successful and no error occurred. A non-zero value will indicate that the background task has failed.

You should not call this method from the main UI thread with a long timeout period to wait for a background task to complete. Windows messages will not be processed while this method is blocked waiting for the background task to complete, and this can cause your application to appear non-responsive to the end-user. If you have a GUI application and you need to determine if a background task has finished, create a timer to periodically call the **TaskDone** method. When it returns **True** (indicating that the task has completed), you can safely call **TaskWait** to obtain the elapsed time and last error code without blocking the current thread.

See Also

TaskCount Property, TaskList Property, TaskAbort Method, TaskDone 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. 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.

The **Reset** and **Uninitialize** methods will abort all active background transfers and wait for those tasks to complete before returning to the caller. It is recommended that your application explicitly wait for background transfers to complete or abort them using this method before allowing the program to terminate. This will ensure that your program can perform any necessary cleanup operations. If there are active background tasks running at the time that the control instance is destroyed, it can force the control to stop those worker threads immediately without waiting for them to terminate gracefully.

See Also

Initialize Method

Write Method

Write data to the server.

Syntax

object.**Write**(*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, you should use a **Byte** array 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 server, or -1 if an error was encountered.

Remarks

The **Write** method sends the data in ***buffer*** to the server. 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 blocking, the application will wait until the data can be sent. If the control is non-blocking and the write fails because it could not send all of the data to the server, the **OnWrite** event will be fired when the server can accept data again.

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 UTF-8 (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

IsConnected Property, IsWritable Property, Timeout Property, Read Method, OnWrite Event

Hypertext Transfer Protocol Control Events

Event	Description
OnCancel	This event is generated when a blocking operation is canceled
OnCommand	This event is generated when the server processes a command issued by the client
OnConnect	This event is generated when a connection is established
OnDisconnect	This event is generated when a connection is terminated
OnError	This event is generated when a control error occurs
OnProgress	This event is generated during data transfer
OnRead	This event is generated when data is available to be read
OnRedirect	This event is generated when the server indicates a resource has been moved
OnTaskBegin	This event is generated when a background task begins
OnTaskEnd	This event is generated when a background task completes
OnTaskRun	This event is generated while a background task is active
OnTimeout	This event is generated when a blocking operation times out
OnWrite	This event is generated when data can be written to the server

OnCancel Event

The **OnCancel** event is generated when a blocking operation is canceled.

Syntax

Sub *object*_**OnCancel** ([*Index* **As Integer**])

Remarks

This event is generated 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, OnTimeout Event

OnCommand Event

The **OnCommand** event is generated when the client sends a command to the server and receives a reply indicating the results of that command.

Syntax

Sub *object*_**OnCommand**([*Index* **As Integer**], **ByVal** *ResultCode* **As Variant**, **ByVal** *ResultString* **As Variant**)

Remarks

The **OnCommand** event is generated when the client receives a reply from the server after some action has been taken. The ***ResultCode*** argument contains the numeric result code returned by the server. The result codes returned from the server fall into one of the following categories:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

The ***ResultString*** argument contains the descriptive string returned by the server which describes the result. The string contents may vary depending on the type of server.

See Also

ResultCode Property, ResultString Property, Command Method

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 server as a result of a **Connect** method call. This event is only triggered when the **Blocking** property is set to False.

See Also

Blocking Property, Connect Method, OnDisconnect Event, OnWrite 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. This event is only triggered when the **Blocking** property is set to False.

When the **OnDisconnect** event fires, it is possible that there may still be buffered data available to read from the server. Before disconnecting from the server, the application should attempt to read any remaining data until the **Read** method returns a value of zero, or returns an error indicating that the operation would block.

See Also

Blocking Property, IsConnected Property, IsReadable Property, Connect Method, Disconnect Method, Read Method, 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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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 during data transfer.

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 during the transfer of data between the client and server, indicating the amount of data exchanged. For transfers of 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

A value that specifies the total amount of data being transferred in bytes. This value may be zero if the control cannot determine the total amount of data that will be copied. If the total number of bytes is less than 2 GiB, the value will be a **Long** (32-bit) integer. For very large transfers, it will be a **Double** floating-point value.

BytesCopied

A value that specifies the number of bytes that have been transferred between the client and server. If the number of bytes copied is less than 2 GiB, the value will be a **Long** (32-bit) integer. For very large transfers, it will be a **Double** floating-point value.

Percent

The percentage of data that's been transferred, expressed as an integer value between 0 and 100, inclusive. If the size of the file on the server cannot be determined, this value will always be 100.

This event is only generated when data is transferred using the **GetData**, **GetFile**, **PostData**, **PostFile**, **PutData** or **PutFile** methods. If the client is reading or writing the file data directly to the server using the **Read** or **Write** methods then the application is responsible for calculating the completion percentage and updating any user interface controls.

See Also

TransferBytes Property, TransferRate Property, TransferTime Property, GetData Method, GetFile Method, PostData Method, PostFile Method, PutData Method, PutFile 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 triggered when the **Blocking** property is set to False.

See Also

IsReadable Property, Read Method, Write Method, OnWrite Event

OnRedirect Event

The **OnRedirect** event is generated when the server indicates a resource has been moved

Syntax

Sub *object*_**OnRedirect** ([*Index* **As Integer**,] **ByVal** *Resource* **As Variant**)

Remarks

This event is generated when the server indicates that the requested resource has been moved to a new location. This new location is typically on the same server, however it may specify another server. The ***Resource*** argument specifies the new location.

If the **AutoRedirect** property is set to True, then the control will automatically retrieve the resource from its new location. If the property is set to False, then the application is responsible for handling the redirection.

See Also

AutoRedirect Property

OnTaskBegin Event

The **OnTaskBegin** event occurs when a background task starts.

Syntax

Sub *object*_**OnTaskBegin** ([*Index* **As Integer**], **ByVal** *TaskId* **As Variant**)

Remarks

The **OnTaskBegin** event is generated when a background task associated with an asynchronous file transfer begins running. The arguments to this event are:

TaskId

An integer value that uniquely identifies the background task.

This event can be used in conjunction with the **OnTaskEnd** event to monitor one or more background tasks that are created to perform asynchronous file transfers. The task ID passed to this event can be used to uniquely identify the task and corresponds to the worker thread that has been created to manage the client session. The application should consider the ID to be an opaque value and never make assumptions about how an ID is assigned to a background task.

See Also

AsyncGetFile Method, AsyncPutFile Method, OnTaskEnd Event, OnTaskRun Event

OnTaskEnd Event

The **OnTaskEnd** event occurs when a background task completes.

Syntax

Sub *object*_**OnTaskEnd** ([*Index* **As Integer**], **ByVal** *TaskId* **As Variant**, **ByVal** *TimeElapsed* **As Variant**, **ByVal** *ErrorCode* **As Variant**)

Remarks

The **OnTaskEnd** event is generated when a file transfer completes and the background task has terminated. The arguments to this event are:

TaskId

An integer value that uniquely identifies the background task.

TimeElapsed

An integer value that specifies the amount of elapsed time in milliseconds.

ErrorCode

An integer value that specifies the last error code for the task.

This event can be used in conjunction with the **OnTaskBegin** event to monitor one or more background tasks that are created to perform asynchronous file transfers. The ***TimeElapsed*** parameter will specify the number of milliseconds that the background task was active. The ***ErrorCode*** parameter specifies the last error code associated with the background task. If this value is zero, that indicates that the task completed successfully. A non-zero value indicates that the task failed and the error code value identifies why the task failed.

See Also

AsyncGetFile Method, AsyncPutFile Method, OnTaskBegin Event, OnTaskRun Event

OnTaskRun Event

The **OnTaskRun** event occurs while a background task is active.

Syntax

Sub *object*_**OnTaskRun** ([*Index* **As Integer**], **ByVal** *TaskId* **As Variant**, **ByVal** *TimeElapsed* **As Variant**, **ByVal** *Completed* **As Variant**)

Remarks

The **OnTaskRun** event is generated periodically during a file transfer while the background task is active. The arguments to this event are:

TaskId

An integer value that uniquely identifies the background task.

TimeElapsed

An integer value that specifies the amount of elapsed time in milliseconds.

Completed

An integer value that specifies an estimated percentage of completion.

The rate and number of times that this event will be generated depends on the task being performed. This event is generally analogous to the **OnProgress** event for file transfers that are performed in the current working thread, however the **OnTaskRun** event will occur for each individual background task that is active. The ***TimeElapsed*** parameter specifies the amount of time that the task has been active, and the ***Completed*** parameter specifies an estimated percentage of completion. This can be used to update the user interface if needed, however it is the application's responsibility to determine which UI component (such as a **ProgressBar** control) is associated with a particular task.

See Also

AsyncGetFile Method, AsyncPutFile Method, OnTaskBegin Event, OnTaskEnd 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. This event is only triggered when the **Blocking** property is set to True.

See Also

Timeout Property, OnCancel Event

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 triggered when the **Blocking** property is set to False.

See Also

IsWritable Property, Read Method, Write Method, OnConnect Event, OnRead Event

Hypertext Transfer Server Control

Implements a server that enables the application to send and receive files using the Hypertext Transfer Protocol.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	HttpServerCtl.HttpServer
File Name	CSHTSX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.HttpServer.11
ClassID	95A2302E-2F22-49AC-A158-5D175C89C64C
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 1945, RFC 2616, RFC 3875

Overview

This ActiveX control provides an interface for implementing an embedded, lightweight server that can be used to provide access to documents and other resources using the Hypertext Transfer Protocol. The server can accept connections from any standard web browser, third-party applications or programs developed using the SocketTools HTTP client API.

The application specifies an initial server configuration and then responds to events that are generated when the client sends a request to the server. An application may implement only minimal handlers for most events, in which case the default actions are performed for most standard HTTP commands. However, an application may also use the event mechanism to filter specific commands or to extend the protocol by providing custom implementations of existing commands or add entirely new commands.

The server includes support for CGI scripting, virtual hosting, client authentication and the creation of virtual directories and files. The server also supports secure connections using TLS. Secure connections require that a valid TLS certificate be installed on the system.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop

and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

Hypertext Transfer 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
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
ClientAccess	Gets and sets the access rights that have been granted to the client session
ClientAddress	Return the Internet address of the current client connection
ClientCount	Return the number of active client sessions connected to the server
ClientHost	Return the host name that the client used to establish the connection
ClientId	Return the unique identifier for the active client session
ClientIdle	Get and set the idle timeout period for the active client session
ClientPort	Return the port number allocated by the active client connection
ClientUser	Return the user name associated with the specified client session
CommandLine	Return the complete command line issued by the client
Directory	Get and set the full path to the root directory assigned to the server
ExecTime	Get and set maximum number of seconds that the server will permit an external command to execute
ExternalAddress	Return the external IP address for the local system
HiddenFiles	Determine if the server should permit access to hidden files
Identity	Gets and sets a string that identifies the server to the client
IdleTime	Gets and sets the maximum number of seconds a client can be idle before the server terminates the session
IsActive	Determine if the server has been started
IsAuthenticated	Determine if the active client session has been authenticated
IsInitialized	Determine if the server has been initialized
IsListening	Determine if the server is listening for client connections
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error that occurred
LocalPath	Return the full path to the local file or directory that is the target of the current request
LocalUser	Determines if the server should perform user authentication using the Windows local account database
LockFiles	Determines if files should be exclusively locked when a client attempts to upload or download a file
LogFile	Gets and sets the name of the server log file
LogFormat	Gets and sets the format used when updating the server log file
LogLevel	Gets and sets the level of detail included in the server log file
MaxClients	Gets and sets the maximum number of clients that are permitted to connect to the server
MemoryUsage	Gets the amount of memory allocated for the server and all client sessions
MultiUser	Determine if the server should be started in multi-user mode
NoIndex	Determine if the server should search for a default index page

Options	Gets and sets the options used when creating an instance of the server
Priority	Gets and sets the priority assigned to the server
ReadOnly	Determine if the server should prevent clients from uploading files
Restricted	Determine if the server should be started in restricted mode, limiting client access to the server
Secure	Determine if the server should accept secure client connections
ServerAddress	Gets and sets the address that will be used by the server to listen for connections
ServerName	Gets and sets the fully qualified domain name for the server
ServerPort	Gets and sets the port number that will be used by the server to listen for connections
ServerUuid	Gets and sets the Universally Unique Identifier (UUID) associated with the server
StackSize	Gets and sets the size of the stack allocated for threads created by the server
ThrowError	Enable or disable error handling by the container of the control
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 object
VirtualPath	Return the virtual path to the local file or directory that is the target of the current command

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, ServerAddress Property, ServerName Property, ServerPort 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 returned IP address allocated for that connection.

Data Type

Integer (Int32)

See Also

AdapterAddress Property, ServerAddress Property, ServerName Property

CertificateName Property

Gets and sets the common name for the server certificate.

Syntax

object.**CertificateName** [= *name*]

Remarks

The **CertificateName** property sets the common name or friendly name of the server certificate that should be used with secure TLS connections. The certificate must be designated as a server certificate and have a private key associated with it, otherwise the server will be unable to create the security context for the client session. This property value is only used if security has been enabled by setting the **Secure** property to **True**.

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, ServerName 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 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

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. 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

ClientAccess Property

Gets and sets the access rights that have been granted to the client session.

Syntax

object.**ClientAccess** [= *accessflags*]

Remarks

The **ClientAccess** property is used to determine all of the access permissions that are currently granted to an authenticated client session and optionally change those permissions. For a list of user access rights that can be granted to the client, see User and File Access Constants.

When modifying the value of this property, it is recommended that you use bitwise OR and AND operands to set and clear specific bit flags. The exception is when using the **httpAccessDefault** permission. If you wish to reset the client session to use the default permissions based on the server configuration and client authentication, then you should assign this value directly to the **ClientAccess** property.

This property should only be accessed within an event handler such as **OnCommand** because its value is specific to the client session that raised the event. This property will always return a value of zero outside of an event handler, and an exception will be raised if you attempt to modify this property outside of an event handler.

Data Type

Integer (Int32)

Example

```
' Allow the client to execute registered programs
HttpServer1.ClientAccess = HttpServer1.ClientAccess Or httpAccessExecute

' Prevent the client from listing files
HttpServer1.ClientAccess = HttpServer1.ClientAccess And Not httpAccessList
```

See Also

AddUser Method, Authenticate Method, OnAuthenticate Event

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 should only be accessed within an event handler such as **OnConnect** because its value is specific to the client session that raised the event. This property will always return an empty string when accessed outside of an event handler.

Data Type

String

See Also

ClientHost Property, ClientPort Property, ServerAddress Property, 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 includes both authenticated and unauthenticated client sessions.

Data Type

Integer (Int32)

See Also

MaxClients Property

ClientHost Property

Return the host name that the client used to establish the connection.

Syntax

object.**ClientHost**

Remarks

The **ClientHost** property returns the host name that the client used to establish the connection. If the client uses HTTP 1.0 or a later version of the protocol, this property will return the host name specified in the request header, otherwise it will return the default host name that was assigned to the server when it started.

Data Type

String

See Also

ClientAddress Property, ClientPort Property, ServerName Property, OnConnect Event

ClientId Property

Return the unique identifier for the active client session.

Syntax

object.**ClientId**

Remarks

Each client connection that is accepted by the server is assigned a unique numeric value. This value is by the application to identify that client session, and is different than the socket handle allocated for the client. Client IDs are unique throughout the life of the server session and are never duplicated.

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. This property will always return a value of zero when accessed outside of an event handler.

Data Type

Integer (Int32)

See Also

ClientAddress Property, ClientHost Property, ServerAddress Property, ServerPort Property

ClientIdle Property

Gets and sets the maximum number of seconds a client can be idle before the server terminates the session.

Syntax

object.**ClientIdle** [= *seconds*]

Remarks

The **ClientIdle** property returns the maximum number of seconds that the active client session may be idle before the server closes the control connection. The idle timeout period for each client session is based on the value of the **IdleTime** property when the server was started, with the default value of 60 seconds. Changing this value inside an event handler will change the timeout period for the active client session.

This property should only be accessed within an event handler such as **OnConnect** because its value is specific to the client session that raised the event. This property will always return a value of zero outside of an event handler, and an exception will be raised if you attempt to modify this property outside of an event handler.

When the timeout period for the client has elapsed, the **OnTimeout** event will fire prior to the client being disconnected from the server.

Data Type

Integer (Int32)

See Also

IdleTime Property, OnTimeout Event

ClientPort Property

Return the port number allocated by the active 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

ClientUser Property

Return the user name associated with the specified client session.

Syntax

object.**ClientUser**

Remarks

The **ClientUser** property returns the user name that the client used to authenticate the client session. This property should only be accessed within an event handler after the client session has been authenticated. Unauthenticated clients are not assigned a user name. This property will always return an empty string when accessed outside of an event handler.

Data Type

String

See Also

ClientAddress Property, Authenticate Method, OnAuthenticate Event

CommandLine Property

Return the complete command line issued by the client.

Syntax

object.**CommandLine**

Remarks

The **CommandLine** property is used to obtain the command that was issued by the client, and is commonly used inside **OnCommand** and **OnResult** event handlers to pre-process and post-process client commands, respectively. If the command sent by the client is used to perform an action on a file or directory, use the **LocalPath** property to get the full path to the local file that is the target of the command. Note that this property only returns the command, and not the associated request headers.

This property should only be accessed within an event handler because its value is specific to the client session that raised the event. This property will always return an empty string when accessed outside of an event handler.

Data Type

String

See Also

LocalPath Property, VirtualPath Property, OnCommand Event, OnResult Event

Directory Property

Get and set the full path to the root directory assigned to the server.

Syntax

object.**Directory** [= *pathname*]

Remarks

The **Directory** property returns the path to the root directory for the server. If this property is set to the name of a valid directory before the server is started, that directory will be considered the root directory for the server. If this property is not set, or is set to an empty string, then the server will use the current working directory as its root directory, however this is not recommended. It is recommended that you specify an absolute path to the directory, otherwise the path will be relative to the current working directory. You may include environment variables in the path surrounded by percent (%) symbols and they will be expanded.

If you have configured the server to permit clients to upload files, you must ensure that your application has permission to create files in the directory that you specify. A recommended location for the server root directory would be a subdirectory of the %ALLUSERSPROFILE% directory. Using the environment variable ensures that your server will work correctly on different versions of Windows. If the root directory does not exist at the time that the server is started, it will be created.

If the **MultiUser** property is **False**, all authenticated clients will have their current working directory initialized to the server root directory. If the **MultiUser** property is **True**, then users are assigned their own home directories and clients can access documents in those directories by including the username in the request URI.

This property can be read after the server has started and it will return the full path to the root directory. However, attempting to change the value of this property after the server has started will cause an exception to be raised. To change the root directory for the server, you must first call the **Stop** method which will terminate all active client connections.

Data Type

String

Example

```
' Set the server root directory
HttpServer1.Directory = "%ALLUSERSPROFILE%\MyProgram\WebServer"
```

See Also

MultiUser Property, AddUser Method

ExecTime Property

Get and set maximum number of seconds that the server will permit an external command to execute.

Syntax

object.**ExecTime** [= *seconds*]

Remarks

The **ExecTime** property specifies the maximum number of seconds that an external program is permitted to run on the server. External programs are either registered using the **RegisterProgram** method, or a script handler is registered using the **RegisterHandler** method. If this value is zero, the default timeout period of 5 seconds will be used. The minimum execution time is 1 second and the maximum time limit is 30 seconds.

Data Type

Integer (Int32)

See Also

IdleTime Property, RegisterHandler Method, RegisterProgram Method, OnExecute Event

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 **ServerAddress** property will only return the IP address for the local system on the LAN side of the network. 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. Note that you should not assign the **ServerAddress** 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 LAN.

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 persistent broadband connections.

Data Type

String

See Also

ClientAddress Property, ServerAddress Property

HiddenFiles Property

Determine if the server should permit access to hidden files.

Syntax

object.**HiddenFiles** [= { True | False }]

Remarks

The **HiddenFiles** property determines if the server should allow clients to access files with the hidden and/or system attribute. If this property is **True**, then hidden files are included in directory listings and clients may download or replace hidden files. If the property is **False**, hidden files are not included in directory listings and any attempt to access, delete or modify a hidden file will result in an error.

The default value for this property is **False**.

Data Type

Boolean

See Also

NoIndex Property, ReadOnly Property, Restricted Property, Start Method

Identity Property

Gets and sets a string that identifies the server to the client.

Syntax

object.**Identity** [= *description*]

Remarks

The **Identity** property returns a string that is used to identify the server. It is used for informational purposes only and does not affect the operation of the server. Typically the string specifies the name of the application and a version number, and is returned to the client as part of the standard response header block. This property can be set to assign an identity to the server, however after the server has started this property becomes read-only.

Data Type

String

See Also

ClientAddress Property, ClientPort Property, ServerName Property, OnConnect Event

IdleTime Property

Gets and sets the maximum number of seconds a client can be idle before the server terminates the session.

Syntax

object.**IdleTime** [= *seconds*]

Remarks

The **IdleTime** property specifies the maximum number of seconds that a client session may be idle before the server closes the control connection to the client. A value of zero specifies the default value of 60 seconds. If the value is non-zero, the minimum value is 10 seconds and the maximum value is 300 seconds (5 minutes). This value is used to initialize the default idle timeout period for each client session. The server determines if a client is idle based on the time the last command was issued and whether or not a data transfer is in progress.

The **ClientIdle** property can be used to determine the idle timeout period for a specific client. When the timeout period for the client has elapsed, the **OnTimeout** event will fire prior to the client being disconnected from the server.

Data Type

Integer (Int32)

See Also

ClientIdle Property, OnTimeout Event

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

IsAuthenticated Property

Determine if the active client session has been authenticated.

Syntax

object.**IsAuthenticated**

Remarks

The **IsAuthenticated** property returns **True** if the active client session has successfully authenticated with a valid username and password. This property should only be accessed within an event handler such as **OnCommand** because its value is specific to the client session that raised the event. This property will always return a value of **False** outside of an event handler.

Data Type

Boolean

See Also

IsListening Property, Authenticate Method, OnAuthenticate Event

IsInitialized Property

Determine if the server has been initialized.

Syntax

object.**IsInitialized**

Remarks

The **IsInitialized** property is used to determine if the current instance of the server 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, Start Method, Stop 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.

Data Type

Boolean

See Also

IsActive Property, Start Method, Stop Method

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

LocalPath Property

Return the full path to the local file or directory that is the target of the current request.

Syntax

object.**LocalPath** [= *filename*]

Remarks

The **LocalPath** property returns the full path to a local file name or directory that maps to the request URI provided by the client. For example, if the client sends the GET command to the server, this property will return the complete path to the local file that the client wants to download. This property will only return a value for those standard HTTP requests that perform some action on a file or directory, otherwise it will return an empty string.

Setting this property allows you to effectively redirect the client to use a different file than the one that was actually requested. If the path is absolute, then it will be used as-is. If the path is relative, it will be relative to the server root directory. The full path to this file is not limited to the server root directory or its subdirectory, it can specify a file anywhere on the local system. If this property is set to an empty string, then the server will revert to using the actual file or directory name specified by the command.

This property should only be set within an **OnCommand** event handler, and only for those requests that perform an action on a file or directory. If the current request does not target a file or directory, setting this property will cause an exception to be raised by the control. Exercise caution when using this property to redirect the server to use a different file than the one requested by the client; changing the target file may cause the client to behave in unexpected ways.

Data Type

String

See Also

VirtualPath Property, ResolvePath Method, OnCommand Event

LocalUser Property

Determines if the server should perform user authentication using the Windows local account database.

Syntax

object.**LocalUser** [= { True | False }]

Remarks

The **LocalUser** property determines if the server should perform user authentication using the Windows local account database. If this option is not specified, the application is responsible for creating virtual users using the **AddUser** method or implementing an **OnAuthenticate** event handler and authenticating client sessions individually.

If this property is set to **True**, a client can authenticate as a local user, however the session will not inherit that user's access rights. All files that are accessed or created by the server will continue to use the permissions of the process that started the server. For example, consider a server application that was started by local user ***A***. Next, a client connects to the server and authenticates itself as local user ***B***. When that client uploads a file to the server using the PUT command, the file that is created will be owned by user ***A***, not user ***B***. This ensures that the server application retains ownership and control of the files that have been created or modified.

The default value for this property is **False**.

Data Type

Boolean

See Also

IsAuthenticated Property, AddUser Method, Authenticate Method, OnAuthenticate Event

LockFiles Property

Determines if files should be exclusively locked when a client attempts to upload or download a file.

Syntax

object.**LockFiles** [= { True | False }]

Remarks

The **LocalTime** property determines if files should be exclusively locked when a client attempts to upload or download a file. If another client attempts to access the same file, the operation will fail. By default, the server will permit multiple clients to access the same file, although it will still write-lock files that are in the process of being uploaded..

The default value for this property is **False**.

Data Type

Boolean

See Also

HiddenFiles Property, ReadOnly Property

LogFile Property

Gets and sets the name of the server log file.

Syntax

object.**LogFile** [= *filename*]

Remarks

The **LogFile** property is used to specify the name of a file that will contain a log of all client activity. The **LogFormat** and **LogLevel** properties affect the specific format for the file and the level of detail included in the log. It is recommended that you specify an absolute path to the log file, otherwise the path will be relative to the current working directory. You may include environment variables in the path surrounded by percent (%) symbols and they will be expanded.

If the log file does not exist it will be created when the server is started. If file already exists, the server will append the new logging data to the file. The server must have permission to create and/or modify the specified file.

Setting this property to an empty string after the server has been started will have the effect of disabling logging, setting the logging level to 0 and the logging format to **httpLogNone**.

Data Type

String

Example

```
' Enable server logging
HttpServer1.LogFile = "%ALLUSERSPROFILE%\MyProgram\WebServer.log"
HttpServer1.LogFormat = httpLogCombined
HttpServer1.LogLevel = 5
```

See Also

LogFormat Property, LogLevel Property

LogFormat Property

Gets and sets the format used when updating the server log file.

Syntax

object.**LogFormat** [= *format*]

Remarks

The **LogFormat** property is used to specify the format of the server log file. It may be one of the following values:

Value	Description
httpLogNone	This value specifies that the server should not create or update a log file. This is the default property value.
httpLogCommon	This value specifies that the log file should use the common log format that records a subset of information in a fixed format. This log format usually only provides information about GET, PUT and POST requests.
httpLogCombined	This value specifies that the server should use the combined log file format. This format is similar to the common format, however it includes the client referrer and user agent. This is the format that most Apache web servers use by default.
httpLogExtended	This value specifies that the log file should use the standard W3C extended log file format. This is an extensible format that can provide additional information about the client session. This format typically generates the largest logfiles.

Data Type

Integer (Int32)

Example

```
' Enable server logging
HttpServer1.LogFile = "%ALLUSERSPROFILE%\MyProgram\Server.log"
HttpServer1.LogFormat = httpLogExtended
HttpServer1.LogLevel = 5
```

See Also

LogFile Property, LogLevel Property

LogLevel Property

Gets and sets the level of detail included in the server log file.

Syntax

object.**LogLevel** [= *level*]

Remarks

The **LogLevel** property is used to specify the level of detail that should generated in the log file. The minimum value is 1 and the maximum value is 10. If this parameter is zero, it is the same as specifying a log file format of **httpLogNone** and will disable logging by the server

Data Type

Integer (Int32)

Example

```
' Enable server logging
HttpServer1.LogFile = "%ALLUSERSPROFILE%\MyProgram\WebServer.log"
HttpServer1.LogFormat = httpLogCombined
HttpServer1.LogLevel = 5
```

See Also

LogFile Property, LogFormat Property

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.

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. You can also use the **Throttle** method to change the maximum number of guest users, the maximum number of clients per IP address and the rate at which clients can connect to 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.

The default value for this property is 100 active client connections.

Data Type

Integer (Int32)

See Also

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

MultiUser Property

Determine if the server should be started in multi-user mode.

Syntax

object.**MultiUser** [= { True | False }]

Remarks

The **MultiUser** property determines if the server should be started in multi-user mode. If this property is set to **True**, virtual users can be assigned their own home directories and clients can access documents in those directories by including the username in the request URI. If this property is set to **False**, all users will share the server root directory by default. This property does not have any effect on the maximum number of simultaneous client sessions that can be established with the server.

Attempting to change the value of this property after the server has started will cause an exception to be raised. To change this property value, you must first call the **Stop** method which will terminate all active client connections.

The default value for this property is **False**.

Data Type

Boolean

See Also

Directory Property, ReadOnly Property, Restricted Property, Start Method

NoIndex Property

Determine if the server should search for a default index page.

Syntax

object.**NoIndex** [= { True | False }]

Remarks

The **NoIndex** property determines if the server should should search for a default index file if the client requests a resource that maps to a local directory on the server. If this property is set to **True**, the server will not search for an index file. If this property is set to **False**, the server will search for a file named index.htm, index.html, default.htm, default.html or index.txt in the directory. If a file by one of those names is found, it will return the contents of that file rather than a list of files in the directory.

Attempting to change the value of this property after the server has started will cause an exception to be raised. To change this property value, you must first call the **Stop** method which will terminate all active client connections.

The default value for this property is **False**.

Data Type

Boolean

See Also

HiddenFiles Property, ReadOnly Property, Restricted Property, Start Method

Options Property

Gets and sets the options used when creating an instance of the server.

Syntax

object.**Options** [= *value*]

Remarks

The **Options** property is an integer value which specifies one or more options. The value specified for this property will be used as the default options when starting the server. The property value is created by using a bitwise operator with one or more of the following values:

Value	Description
httpServerMultiUser	This option specifies the server should be started in multi-user mode, where users are assigned their own home directories and clients can access documents in those directories by including the username in the request URI. If this option is not specified, then all users will share the server root directory by default. This option does not have any effect on the maximum number of simultaneous client sessions that can be established with the server.
httpServerRestricted	This option specifies the server should be initialized in a restricted mode, limiting certain functionality. The only commands accepted by the server will be the GET and HEAD commands. The server will never return a list of files if the client provides a URL that maps to a local directory and there is no default index page. Clients will not be able to execute CGI programs or scripts, and cannot access files outside of the server root directory or its subdirectories.
httpServerLocalUser	This option specifies the server should perform user authentication using the Windows local account database. This option is useful if the server should accept local usernames, or if the application does not wish to implement an event handler for user authentication. If this option is not specified, the application is responsible for authenticating all users.
httpServerNoIndex	This option specifies the server should not search for a default index page if the client provides a URL that maps to a local directory. By default, the server will search for a file named index.htm, index.html, default.htm, default.html or index.txt in the directory. If a file by one of those names is found, it will return the contents of that file rather than a list of files in the directory.
httpServerReadOnly	This option specifies the server should only allow read-only access to files by default. If this option is enabled, it will change the default permissions granted to authenticated users. Commands that are used to create, delete or modify files on the server will be disabled by default. It is recommended that this option be enabled if the server is publicly accessible over the

	Internet.
httpServerLockFiles	This option specifies that files should be exclusively locked when a client attempts to upload or download a file. If another client attempts to access the same file, the operation will fail. By default, the server will permit multiple clients to access the same file, although it will still write-lock files that are in the process of being uploaded.
httpServerHiddenFiles	This option specifies that when a client requests a resource, the server should permit access to hidden and system files or subdirectories. By default, the server will not allow access to a hidden or system file, even if the client session has been authenticated. This option is ignored if the server is started in restricted mode.
httpServerSecure	This option specifies that secure connections using TLS should be enabled. This option requires that a valid TLS certificate be installed on the local host. The default port number for secure HTTP connections is 443. If security is enabled, all client connections to the server must be secure. Standard and secure connections cannot be shared by the same instance of the server. If your application must support both standard and secure connections, you must create two instances of the server listening on two different ports, one with the **httpServerSecure** option enabled and the other without.
httpServerSecureFallback	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.

Most of these options have a corresponding Boolean property. For example, the **httpServerRestricted** option corresponds to the **Restricted** property, where setting the property to True enables the option and setting it to False disables the option.

In most cases, it is recommended that you use the property value related to the option, rather than setting the **Options** property. It will make your code more readable and prevent potential compatibility issues with subsequent versions of the control. If you do decide to specify option bit flags, it is recommended that you use the constant name rather than the numeric value.

Data Type

Integer (Int32)

See Also

HiddenFiles Property, LocalUser Property, LockFiles Property, MultiUser Property, ReadOnly Property, Restricted Property, Secure Property, Start Method

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
httpPriorityBackground	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.
httpPriorityLow	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.
httpPriorityNormal	The default priority which balances resource and processor utilization. It is recommended that most applications use this priority.
httpPriorityHigh	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.
httpPriorityCritical	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 **httpPriorityNormal** 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

Start Method

ReadOnly Property

Determine if the server should prevent clients from uploading files.

Syntax

object.**ReadOnly** [= { True | False }]

Remarks

The **ReadOnly** property determines if the server should only allow read-only access to files by default, changing the default permissions granted to authenticated users. If this property is set to **True**, commands that are used to create, delete or modify files on the server will be disabled by default.

Attempting to change the value of this property after the server has started will cause an exception to be raised. To change this property value, you must first call the **Stop** method which will terminate all active client connections.

The default value for this property is **False**.

Data Type

Boolean

See Also

Directory Property, NoIndex Property, ReadOnly Property, Restricted Property, Start Method

Restricted Property

Determine if the server should be started in restricted mode, limiting client access to the server.

Syntax

object.**Restricted** [= { True | False }]

Remarks

The **Restricted** property determines if the server should be initialized in a restricted mode that isolates the server and limits the ability for clients to access files on the host system. If this property is set to **True**, the only commands accepted by the server will be the GET and HEAD commands. The server will never return a list of files if the client provides a URL that maps to a local directory and there is no default index page. Clients will not be able to execute CGI programs or scripts, and cannot access files outside of the server root directory or its subdirectories.

Attempting to change the value of this property after the server has started will cause an exception to be raised. To change this property value, you must first call the **Stop** method which will terminate all active client connections.

The default value for this property is **False**.

Data Type

Boolean

See Also

Directory Property, MultiUser Property, NoIndex Property, Restricted Property, Start 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.

Data Type

Boolean

See Also

CertificateName Property, CertificateStore 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. By default the server will accept connections on any appropriately configured network adapter. 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.

If an IPv6 address is specified as the server 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.

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

Gets and sets the fully qualified domain name for the server.

Syntax

object.**ServerName** [= *hostname*]

Remarks

The **ServerName** property returns the fully qualified domain name assigned to the server. 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.

Setting this property assigns the default hostname for the server which is returned to the client in the standard response header block. If the server is publicly accessible over the Internet, this property should be set to the same hostname that is associated with the server IP address.

Attempting to change the value of this property after the server has started will cause an exception to be raised. To change this property value, you must first call the **Stop** method which will terminate all active client connections.

Data Type

String

See Also

ExternalAddress Property, ServerAddress Property, ServerPort Property, AddHost Method, Start 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. The default port numbers used are port 80 for standard connections and port 443 for secure connections.

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. Attempting to change the value of this property after the server has started will cause an exception to be raised. To change this property value, you must first call the **Stop** method which will terminate all active client connections.

Data Type

Integer (Int32)

See Also

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

ServerUuid Property

Gets and sets the Universally Unique Identifier (UUID) associated with the server.

Syntax

object.**ServerUuid** [= *uuid*]

Remarks

The **ServerUuid** property returns the UUID that uniquely identifies this instance of the server. If the application does not set this property, a temporary UUID will be assigned to the server. If a value is assigned to this property, it must be a valid UUID string. A permanent UUID can be generated using a utility such as **uuidgen** which is included with Visual Studio.

Attempting to change the value of this property after the server has started will cause an exception to be raised. To change this property value, you must first call the **Stop** method which will terminate all active client connections.

Data Type

String

See Also

ServerAddress Property, ServerName Property, ServerPort Property, Start Method

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

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to **True**, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

See Also

LastError Property, OnError Event

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the tracing of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the HTTP client and server controls, and you set the **Trace** property to **True** on the HTTP client control, function calls made by both controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named CSTRACE.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
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).

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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing 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
stTraceInfo	All function calls are written to the trace file. This is the default value.
stTraceError	Only those function calls which fail are recorded in the trace file.
stTraceWarning	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file.
stTraceHexDump	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.

Since socket function tracing is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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 the error WSAEWOULDBLOCK is returned, a warning is generated since the application simply needs to attempt to write the data at a later time.

Data Type

String

See Also

Trace Property, TraceFile Property

Version Property

Return the current version of the object.

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

VirtualPath Property

Return the virtual path to the local file or directory that is the target of the current command.

Syntax

object.**VirtualPath** [= *filename*]

Remarks

The **VirtualPath** property returns the virtual path to the resource requested by the client. For example, if the client sends the GET command to the server, this property will return the complete virtual path to the local resource that the client wants to download. This property will only return a value for those requests that perform some action on a file or directory, otherwise it will return an empty string.

Setting this property allows you to effectively redirect the client to use a different resource than the one that was actually requested. If the path is absolute, then it will be used as-is. If the path is relative, it will be relative to the server root directory. If this property is set to an empty string, then the server will revert to using the actual file or directory name specified by the command.

This property should only be set within an **OnCommand** event handler, and only for those requests that perform an action on a file or directory. If the current command does not target a file or directory, setting this property will cause an exception to be raised by the control. Exercise caution when using this property to redirect the server to use a different file than the one requested by the client; changing the target file may cause the client to behave in unexpected ways.

To instruct the client to use a different resource URI, it is recommended that you use the **RedirectRequest** method rather than modifying the value of this property.

Data Type

String

See Also

LocalPath Property, RedirectRequest Method, ResolvePath Method, OnCommand Event

Hypertext Transfer Server Control Methods

Method	Description
AddHost	Add a new virtual host to the server virtual host table
AddPath	Add a new virtual path for the specified host
AddUser	Add a new virtual user to the server
Authenticate	Authenticate the client and assign access rights for the session
CheckPath	Determine if the client has permission to access the specified virtual path
ClearHeaders	Delete all of the response headers for the specified client session
DeleteHost	Delete a virtual host associated with the specified server
DeletePath	Delete a virtual path from the specified virtual host
DeleteUser	Remove a virtual user from the server
Disconnect	Disconnect the specified client session from the server
GetAllHeaders	Return all of the request header values in the specified string buffer
GetHeader	Return the value of a request header for the specified client session
GetVariable	Return the value of a CGI environment variable for the specified client
Initialize	Initialize the control and validate the runtime license key
ReceiveRequest	Receive the request that was sent by the client to the server
RedirectRequest	Redirect the request from the client to another URL
RegisterHandler	Register a CGI program for use and associate it with a file name extension
RegisterProgram	Register a CGI program for use and associate it with a virtual path on the server
RequireAuthentication	Send a response to the client indicating that authentication is required
Reset	Reset the internal state of the control to its default values
ResolvePath	Resolve a path to its full virtual or local file name
Restart	Restart the server, terminating all active client connections
Resume	Resume accepting new client connections
SendError	Send a customized error response to the specified client
SendResponse	Send a result code and message to the client in response to a command
SetHeader	Create or change the value of a response header for the client session
SetVariable	Create or change the value of a CGI environment variable for the specified client
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

AddHost Method

Add a new virtual host to the server virtual host table.

Syntax

object.**AddHost**(*VirtualHost*, [*VirtualPort*], [*Directory*])

Parameters

VirtualHost

A string which specifies the hostname that will be added to the virtual host table. This parameter must specify a valid hostname and cannot be a zero-length string.

VirtualPort

An optional integer value which specifies the port number for the virtual host. If this parameter is specified, the value must be zero or the same value as the original port number that the server was configured to use. Port-based virtual hosting is currently not supported and this parameter is included for future use.

Directory

An optional string that specifies the root document directory for the virtual host. If this parameter is omitted or a zero-length string, the virtual host will use the same root directory that was specified when the server was started. This parameter may contain environment variables enclosed in % symbols.

Return Value

An integer value that specifies the host ID that is used to reference the virtual host. A return value of -1 indicates that an error has occurred.

Remarks

Virtual hosting is a method for sharing multiple domain names on a single instance of a server. The client provides the server with the hostname that it has used to establish the connection, and that name is compared against a table of virtual hosts configured for the server. If the hostname matches a virtual host, the client will use the root directory and any virtual paths that have been assigned to that host.

When the server is first started, a default virtual host with an ID of zero is automatically created and is identified as **httpHostDefault**. This virtual host uses the same hostname, port number and root directory that the server instance was created with. The application should treat all other host IDs as opaque values and never make assumptions about how they are allocated.

The virtual host ID returned by this method can be used with the **AddPath** method to create a virtual path assigned to the host, the **AddUser** to create a virtual user, and the **RegisterHandler** and **RegisterProgram** methods which are used to register script handlers and CGI programs.

See Also

AddPath Method, AddUser Method, DeleteHost Method, RegisterHandler Method, RegisterProgram Method

AddPath Method

Add a new virtual path for the specified host.

Syntax

object.**AddPath**(*HostId*, *VirtualPath*, *LocalPath*, [*AccessFlags*])

Parameters

HostId

An integer value which identifies the virtual host. A value of zero specifies that the default virtual host should be used.

VirtualPath

A string which specifies the virtual path that will be created. This parameter cannot be an empty string and the maximum length of the virtual path is 1024 characters.

LocalPath

A string which specifies the local directory or file name that the virtual path will be mapped to. This path must exist and can be no longer than 260 characters. This parameter cannot be an empty string.

AccessFlags

An optional integer value which specifies the access clients will be given to the virtual path. This value created from one or more bit flags. For a list of access permissions, see User and File Access Constants. If this parameter is omitted, the virtual path is assigned default file access permissions based on the server configuration.

Return Value

A value of zero is returned if the virtual path was created. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **AddPath** method maps a virtual path name to a directory or file name on the local system. Virtual paths are assigned to specific hosts and if multiple virtual hosts are created for the server, each can have its own virtual paths which map to different files. To create a virtual path for the default server, the caller should specify the ***HostId*** parameter as **httpHostDefault** which has a value of zero.

It is recommended that the ***LocalPath*** parameter always specify the full path to the local file or directory. If the path is relative, it will be considered relative to the current working directory for the process and expanded to its full path name. The local path can include environment variables surrounded by % symbols. For example, if the value %ProgramData% is included in the path, it will be expanded to the full path for the common application data folder. The local path cannot specify a Windows system folder or the root directory of a mounted drive volume.

The local file or directory does not need to located in the document root directory for the server or virtual host. It can specify any valid local path that the server process has the appropriate permissions to access. You should exercise caution when creating virtual paths to files or directories outside of the server root directory. If the ***LocalPath*** parameter specifies a directory, clients will have access to that directory and all subdirectories using its virtual path.

If you wish to password protect the virtual file or directory, include the **httpAccessProtected** flag in the file permissions. The default command handlers will recognize this flag and require that the client authenticate itself to grant access to the resource. If the server application implements a custom

command handler, it is responsible for checking for the presence of this flag and perform the appropriate checks to ensure that the client session has been authenticated.

If the server was started in restricted mode, the client will be unable to access documents outside of the server root directory and its subdirectories. This restriction also applies to virtual paths that reference documents or other resources outside of the root directory. To allow a client to access a document outside of the server root directory, the **ClientAccess** property should be used to grant the client **httpAccessRead** permission.

See Also

ClientAccess Property, LocalPath Property, VirtualPath Property, DeletePath Method, ResolvePath Method

AddUser Method

Add a new virtual user to the server.

Syntax

object.**AddUser**(*HostId*, *UserName*, *Password*, [*AccessFlags*], [*Directory*])

Parameters

HostId

An integer value which identifies the virtual host. A value of zero specifies that the default virtual host should be used.

UserName

A string which specifies the user name. The maximum length of a username is 63 characters and it is recommended that names be limited to alphanumeric characters. Whitespace, control characters and certain symbols such as path delimiters and wildcard characters are not permitted. If an invalid character is included in the name, the method will fail with an error indicating the username is invalid. The username must be at least three characters in length. Usernames are not case sensitive.

Password

A string which specifies the user password. The maximum length of a password is 63 characters and is limited to printable characters. Whitespace and control characters are not permitted. If an invalid character is included in the password, the method will fail with an error indicating the password is invalid. The password must be at least one character in length. Passwords are case sensitive.

AccessFlags

An optional integer value which specifies the access clients will be given when authenticated as this user. This value created from one or more bit flags. For a list of user access permissions, see User and File Access Constants. If this parameter is omitted, the user is assigned default access permissions based on the server configuration.

Directory

An optional string which specifies the directory that will be the virtual user's home directory. If the server was started in multi-user mode, this directory will be relative to the user directory created by the server, otherwise it will be relative to the server root directory. If the directory does not exist, it will be created the first time that the virtual user successfully logs in to the server. If this parameter is omitted or is an empty string, a default home directory will be created for the virtual user.

Return Value

A value of zero is returned if the virtual user was created. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **AddUser** method creates a virtual user that is associated with the server. When a client connects with the server and provides authentication credentials, the server will check if the username has been created using this method. If a match is found, the client access rights will be updated.

If you wish to modify the information for an existing user, it is not necessary to delete the username first. If this method is called with a username that already exists, that record is replaced with the values passed to this method.

The virtual users created by this method exist only as long as the server is active. If you wish to maintain a persistent database of users and passwords, you are responsible for its implementation

based on the requirements of your specific application. For example, a simple implementation would be to store the user information in a local XML or INI file and then read that configuration file after the server has started, calling this method for each user that is listed.

See Also

Authenticate Method, DeleteUser Method, RequireAuthentication Method, OnAuthenticate Event

Authenticate Method

Authenticate the client and assign access rights for the session.

Syntax

object.**Authenticate**(*ClientId*, [*AccessFlags*])

Parameters

ClientId

An integer that identifies the client session.

AccessFlags

An optional integer value which specifies the access clients will be given when authenticated as this user. This value created from one or more bit flags. For a list of user access permissions, see User and File Access Constants. If this parameter is omitted, the client is authenticated using the default access permissions based on the server configuration.

Return Value

A value of zero is returned if the client session was authenticated. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Authenticate** method is used to authenticate a specific client session, typically in response to an **OnAuthenticate** event that indicates a client has provided authentication credentials as part of the request for a document or other resource.

To enable the server to automatically authenticate a client session, use the **AddUser** method to add one or more virtual users. The server will search the list of virtual users for a match to the credentials provided by the client and will set the appropriate permissions for the session without requiring a event handler to manually authenticate the session using this method.

If the server was started with the **LocalUser** property set to **True** and the client session is not authenticated using this method, the server will attempt to authenticate the client session using the local Windows user database. Although this option can be convenient because it does not require the implementation of an event handler for the **OnAuthenticate** event, it can be used by clients to attempt to discover valid usernames and passwords for the local system. It is recommended that you use the **AddUser** method to create virtual users rather than using the local user database.

See Also

MultiUser Property, Restricted Property, AddUser Method, DeleteUser Method, OnAuthenticate Event

CheckPath Method

Determine if the client has permission to access the specified virtual path.

Syntax

object.**CheckPath**(*ClientId*, *VirtualPath*, [*AccessFlags*])

Parameters

ClientId

An integer value which identifies the client session.

VirtualPath

A string which specifies the virtual path that will be created. The path must be absolute and cannot be an empty string. The maximum length of the virtual path is 1024 characters.

AccessFlags

An optional integer value which specifies the access permissions that should be checked. This value created from one or more bit flags. For a list of access permissions, see User and File Access Constants. If this parameter is omitted, the method checks to ensure the client has read access to the virtual path.

Return Value

A Boolean value that specifies if the client has access to the virtual path. A return value of **True** indicates that the virtual path exists and the client has the requested permissions. A return value of **False** indicates that the path does not exist, or the client does not have the requested access to the file or directory.

Remarks

The **CheckPath** method is used to determine if the client has permission to access the virtual file or directory, based on the value of the ***AccessFlags*** parameter. For example, if the ***AccessFlags*** parameter has the value **httpAccessWrite**, this method will check if the client has write permission for the file or directory. The method will return a non-zero value if the client does have the requested permission, or zero if it does not.

Applications that implement their own custom handlers for standard HTTP commands should use this method to ensure that the client has the appropriate permissions to access the requested resource. Failure to check the access permissions for the client can result in the client being able to access restricted documents and other resources. It is recommended that most applications use the default command handlers.

To obtain the path to the local file or directory that the virtual path is mapped to, use the **ResolvePath** method.

See Also

ClientAccess Property, LocalPath Property, VirtualPath Property, AddPath Method, DeletePath Method, ResolvePath Method

ClearHeaders Method

Delete all of the response headers for the specified client session.

Syntax

object.**ClearHeaders**(*ClientId*)

Parameters

ClientId

An integer value which identifies the client session.

Return Value

A value of zero is returned if the response headers were deleted. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ClearHeaders** method is used to delete all of the current response header values and automatically generate a new set of default response headers. This method can be useful if the client application wants to clear any custom headers that were specified prior to sending a response to the client. In most cases it is not necessary to use this method because the server will automatically clear the response headers when a session terminates.

See Also

GetHeader Method, SetHeader Method

DeleteHost Method

Delete a virtual host associated with the specified server.

Syntax

object.**DeleteHost**(*HostId*)

Parameters

HostId

An integer value which identifies the virtual host.

Return Value

A value of zero is returned if the virtual host was deleted. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **DeleteHost** method removes a virtual host that was created by a previous call to the **AddHost** method. All virtual paths and users associated with the specified host are no longer valid. It is not necessary to call this method to delete any of the virtual hosts prior to stopping the server. Part of the normal shutdown process is releasing the resources allocated for each virtual host that was added to the server.

This method cannot be used to delete the virtual host with an ID of zero, which is the default virtual host that is allocated when the server is started.

See Also

ServerName Property, AddHost Method

DeletePath Method

Delete a virtual path from the specified virtual host.

Syntax

object.**DeletePath**(*HostId*, *VirtualPath*)

Parameters

HostId

An integer value which identifies the virtual host.

VirtualPath

A string that specifies the virtual path to be removed. This path must be absolute and cannot be an empty string.

Return Value

A value of zero is returned if the virtual path was deleted. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

This method removes a virtual path that was created by a previous call to the **AddPath** method.

See Also

LocalPath Property, VirtualPath Property, AddPath Method, CheckPath Method, ResolvePath Method

DeleteUser Method

Remove a virtual user from the server.

Syntax

object.**DeleteUser**(*HostId*, *UserName*)

Parameters

HostId

An integer value which identifies the virtual host.

UserName

A string which specifies the user name to be deleted. Usernames are not case sensitive.

Return Value

A value of zero is returned if the virtual user was deleted. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **DeleteUser** method removes a virtual user that was created by a previous call to the **AddUser** method. This method will not match partial usernames and wildcard characters cannot be used to delete multiple users. Usernames are not case sensitive.

See Also

AddUser Method, Authenticate Method, OnAuthenticate Event

Disconnect Method

Disconnect the specified client session from the server.

Syntax

object.**Disconnect**(*ClientId*)

Parameters

ClientId

An integer that identifies the client session.

Return Value

A value of zero is returned if the client was signaled to terminate its connection to the server. 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 other resources that were 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 disconnect the client.

The thread that is managing the client will be signaled that it should disconnect from the server, and it will begin the process of terminating the session. This is an asynchronous process and it is not guaranteed that the client will have actually disconnected from the server at the time that this method returns to the caller.

See Also

Start Method, Stop Method, OnConnect Event, OnDisconnect Event

GetAllHeaders Method

Return all of the request header values in the specified string buffer.

Syntax

object.**GetAllHeaders**(*ClientId*, *Headers*)

Parameters

ClientId

An integer that identifies the client session.

Headers

A string variable that is passed by reference which will contain the request headers when the method returns.

Return Value

A Boolean value that specifies if the method was successful.

Remarks

The **GetAllHeaders** method is used to obtain all of the request headers that were provided by the client. Each header name is separated from its value by the colon (:) and each header is terminated with a carriage return and linefeed (CRLF) sequence. Typically this method would be used within an **OnCommand** event handler. To get the value of a specific request header, use the **GetHeader** method.

Refer to Hypertext Transfer Protocol Headers for a list of common request and response headers that are used.

See Also

GetHeader Method, SetHeader Method, OnCommand Event

GetHeader Method

Return the value of a request header for the specified client session.

Syntax

object.**GetHeader**(*ClientId*, *HeaderName*, *HeaderValue*)

Parameters

ClientId

An integer that identifies the client session.

HeaderName

A string that specifies the name of the header field. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value.

HeaderValue

A string variable that is passed by reference which will contain the value of the header when the method returns.

Return Value

A Boolean value that specifies if the method was successful.

Remarks

The **GetHeader** method will return the value of a specific header field included in the request sent by the client. Typically this is used within an **OnCommand** event handler when the server application needs to process a custom command. The **GetAllHeaders** method can be used to obtain a copy of the complete request header block submitted by the client.

Refer to Hypertext Transfer Protocol Headers for a list of common request and response headers that are used.

See Also

GetAllHeaders Method, SetHeader Method, OnCommand Event

GetVariable Method

Return the value of a CGI environment variable for the specified client.

Syntax

object.**GetVariable**(*ClientId*, *VariableName*, *VariableValue*)

Parameters

ClientId

An integer that identifies the client session.

VariableName

A string that that specifies the name of the environment variable. Variable names are not case-sensitive and should not include the equal sign which acts as a delimiter that separates the variable name from its value.

VariableValue

A string variable that is passed by reference which will contain the value of the environment variable when the method returns.

Return Value

A Boolean value that specifies if the method was successful.

Remarks

The **GetVariable** method will return the value of an environment variable that has been defined for the client. Each client session inherits a copy of the process environment block, which is then modified to define various environment variables that are used with CGI programs and scripts. The **SetVariable** method can be used to change existing environment variables or create new variables.

The standard CGI environment variables that are defined by the server are not created until the client request has been processed. This means that environment variables such as REMOTE_ADDR and SERVER_NAME will not be defined inside an **OnConnect** event handler.

See Also

SetVariable Method, OnExecute Event

Initialize Method

Initialize the server 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 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

```
Dim objServer As Object
Set objServer = CreateObject("SocketTools.HttpServer.11")

nError = objServer.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize the SocketTools object"
    End
End If
```

See Also

IsInitialized Property, Start Method, Stop Method, Uninitialize Method

ReceiveRequest Method

Receive the request that was sent by the client to the server.

Syntax

object.**ReceiveRequest**(*ClientId*, *Buffer*, [*Length*])

Parameters

ClientId

An integer that identifies the client session.

Buffer

A string variable or byte array that is passed by reference which will contain any request data submitted by the client.

Length

An optional integer value that specifies the maximum amount of data returned by the method. If this parameter is omitted the entire request buffer will be returned. If a fixed length string or byte array is provided as the buffer, the maximum amount of data returned is limited by the size of the buffer.

Return Value

A value of zero is returned if the method completed successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ReceiveRequest** method is called within an **OnCommand** event handler to process the command issued by the client and return information about the request to the server application. It is only necessary for the application to call this method if it wants to implement its own custom handling for a command.

It is recommended that you only use this method to process custom commands and not standard commands such as GET and POST. This ensures that the appropriate security checks are performed and the response conforms to the protocol standard. After the request data has been processed, the application should use the **SendResponse** or **SendError** method to send a response back to the client indicating success or failure.

This method may only be called once per command issued by the client.

See Also

SendError Method, SendResponse Method, OnCommand Event

RedirectRequest Method

Redirect the request from the client to another URL.

Syntax

object.**RedirectRequest**(*ClientId*, *Location*, [*Method*])

Parameters

ClientId

An integer that identifies the client session.

Location

A string that specifies the new location for the requested resource. This value must be a complete URL, including the http:// or https:// scheme.

Method

An optional integer value that specifies if the redirection is permanent or temporary. If this parameter is omitted, the client will be informed that the redirection is temporary. One of the following values may be used:

Value	Description
httpRedirectPermanent	This value is used for permanent redirection, indicating that the client should update any record of the link with the new URL specified by the ***Location*** parameter. This result is cacheable and when the client makes subsequent requests for the resource, it should always use the new URL.
httpRedirectTemporary	This value is used for temporary redirection, indicating that the client should issue a request for the resource using the new URL specified by the ***Location*** parameter, but subsequent requests should continue to use the original URL.
httpRedirectOther	This value is used for temporary redirection, however it instructs the client that it should use the GET command to request the redirected resource. This option is typically used to redirect a client after it has used the POST command.

Return Value

A value of zero is returned if the method completed successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **RedirectRequest** method can be used within an **OnCommand** event handler to redirect the client to a new location for the resource that it has requested. This redirection can be permanent or temporary, depending on whether the server expects the client to continue to use the original URL when requesting the resource.

If **httpRedirectTemporary** is specified, the actual status code that is returned to the client depends on the version of the protocol that is being used. If the client has issued the request using HTTP 1.0 then the server will return a 302 code to the client. If the client used HTTP 1.1, the server will return a 307 code to the client that indicates it should use the same command verb (GET, POST, etc.) when requesting the resource at the new location.

If **httpRedirectOther** is specified, the status code that is returned to the client depends on which version of the protocol is being used. For clients who are using HTTP 1.0, the server will return a 302 code to the client just as with **httpRedirectTemporary**. If the client is using HTTP 1.1, the server will return a 303 code to the client that indicates it should always use the GET command to request the new resource, regardless if a different command was originally used (POST, PUT, etc.)

This method provides a simplified interface for sending a redirection status code that also implicitly sets the Location response header to the value of the ***Location*** parameter. If the server application needs to send alternate redirection codes such as 305 (Use Proxy) then it should use **SetHeader** method to set the value of the Location response header, followed by the **SendReponse** method to send the redirection status code.

See Also

SendError Method, SendResponse Method, OnCommand Event

RegisterHandler Method

Register a CGI program for use and associate it with a file name extension.

Syntax

object.**RegisterHandler**(*HostId*, *FileExtension*, *ProgramFile*, [*Parameters*], [*Directory*])

Parameters

HostId

An integer value which identifies the virtual host. A value of zero specifies that the default virtual host should be used.

FileExtension

A string which specifies the file name extension that is associated with the CGI program.

ProgramFile

A string which specifies the full path to the CGI program on the local system.

Parameters

An optional string that specifies additional parameters for the program. This value will be passed to the program as command line arguments. If the CGI program does not require any command line parameters, this parameter may be omitted.

Directory

An optional string that specifies the current working directory for the program. If this parameter is omitted, the server will use the root document directory for the virtual host.

Return Value

A value of zero is returned if the program was registered successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **RegisterHandler** method registers an executable CGI program and associates it with a file name extension. When the client issues a GET or POST command that specifies a file with that extension, the program will be executed and the output return to the client.

The ***ProgramFile*** string specifies file name of the CGI program. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the ability to directly access the executable file itself. It is permitted to have multiple file name extensions that reference the same program. The only requirement is that the extension be unique for the given host. The program name may contain environment variables surrounded by % symbols. For example, %ProgramFiles% would be expanded to the `C:\Program Files` folder.

It is important to note that the program specified by ***ProgramFile*** must be an executable file, not a script or batch file. If the program name does not contain a directory path, then the standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The ***Parameters*** string can specify additional command line parameters that should be passed to the CGI program as arguments. This string can also contain a placeholder named "%1" that will be replaced by the full path to the local script filename. If no placeholder is included in the parameters, or the ***Parameters*** argument is omitted, the script file name will be passed to the program as its only argument.

The executable program that is registered using this program must be a console application that conforms to the CGI/1.1 specification defined in RFC 3875. Request data submitted by the client as part of a POST will be provided to the program as standard input. The output from the program must be written to standard output. The first lines of output from the program should be any response headers, followed by an empty line. Each line should be terminated with a carriage-return and linefeed (CRLF) sequence. If the CGI program outputs additional data to be processed by the client, it should provide Content-Type and Content-Length response headers.

When developing a CGI program, it is important to take into consideration the environment that it will be executing in. The program will be started as a child process of the server application, and will inherit the same privileges. This means that it will typically have access to the boot drive, the Windows folders and the system registry. CGI programs must ensure that all query parameters and request data submitted by the client have been validated.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

Example

```
// Register a handler for VBScript
HttpServer1.RegisterHandler httpHostDefault, "vbs",
"%SystemRoot%\System32\cscript.exe", "/nologo /b ""%1"""
```

See Also

RegisterProgram Method, OnCommand Event, OnExecute Event

RegisterProgram Method

Register a CGI program for use and associate it with a virtual path on the server.

Syntax

object.**RegisterProgram**(*HostId*, *CommandName*, *ProgramFile*, [*Parameters*], [*Directory*])

Parameters

HostId

An integer value which identifies the virtual host. A value of zero specifies that the default virtual host should be used.

CommandName

A string which specifies the virtual path to the CGI program. This must be an absolute path, but does not have to specify a pre-existing virtual path or map to the directory structure of the root document directory for the server. The maximum length of the virtual path is 1024 characters.

ProgramFile

A string which specifies the full path to the CGI program on the local system.

Parameters

An optional string that specifies additional parameters for the program. This value will be passed to the program as command line arguments. If the CGI program does not require any command line parameters, this parameter may be omitted.

Directory

An optional string that specifies the current working directory for the program. If this parameter is omitted, the server will use the root document directory for the virtual host.

Return Value

A value of zero is returned if the program was registered successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **RegisterProgram** method registers a CGI program and associates it with a virtual path. When the client issues a GET or POST command specifying the virtual path associated with the program, the program will be executed and the output return to the client.

The ***ProgramFile*** string specifies file name of the CGI program. You should not install any executable programs in the server root directory or its subdirectories. A client should never have the ability to directly access the executable file itself. It is permitted to have multiple virtual paths that reference the same executable file. The only requirement is that the virtual path be unique for the given host. The program name may contain environment variables surrounded by % symbols. For example, %ProgramFiles% would be expanded to the `C:\Program Files` folder.

It is important to note that the program specified by ***ProgramFile*** must be an executable file, not a script or batch file. If the program name does not contain a directory path, then the standard Windows pathing rules will be used when searching for an executable file that matches the given name. It is recommended that you always provide a full path to the executable file.

The ***Parameters*** string can specify additional command line parameters that should be passed to the CGI program as arguments. This string can also contain a placeholder named "%1" that will be replaced by the virtual path associated with the program. If this argument is omitted, no additional parameters are passed to the program.

The executable program that is registered using this program must be a console application that conforms to the CGI/1.1 specification defined in RFC 3875. Request data submitted by the client as part of a POST will be provided to the program as standard input. The output from the program must be written to standard output. The first lines of output from the program should be any response headers, followed by an empty line. Each line should be terminated with a carriage-return and linefeed (CRLF) sequence. If the CGI program outputs additional data to be processed by the client, it should provide Content-Type and Content-Length response headers.

When developing a CGI program, it is important to take into consideration the environment that it will be executing in. The program will be started as a child process of the server application, and will inherit the same privileges. This means that it will typically have access to the boot drive, the Windows folders and the system registry. CGI programs must ensure that all query parameters and request data submitted by the client have been validated.

If the server is running on a system with User Account Control (UAC) enabled and does not have elevated privileges, do not register a program that requires elevated privileges or has a manifest that specifies the requestedExecutionLevel as requiring administrative privileges.

See Also

RegisterHandler Method, OnCommand Event, OnExecute Event

RequireAuthentication Method

Send a response to the client indicating that authentication is required.

Syntax

object.**RequireAuthentication**(*ClientId*, [*AuthType*], [*Realm*])

Parameters

ClientId

An integer that identifies the client session.

AuthType

An optional integer value that corresponds to a result code, informing the client if the redirection is permanent or temporary. If this parameter is omitted, then Basic authentication will be used by default. This parameter may be one of the following values:

Value	Description
httpAuthBasic	This option specifies the Basic authentication scheme should be used. This option is supported by all clients that support at least version 1.0 of the protocol.

Realm

An optional string value that is displayed a web browser to indicate to the user which username and password they should use. If this parameter is omitted or is an empty string, the domain name the client used to establish the connection will be used.

Return Value

A value of zero is returned if the method completed successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **RequireAuthentication** method can be used within an **OnCommand** event handler to indicate to the client that it must provide a username and password to access the requested resource. The client should respond by issuing another request that includes the required credentials. To determine if a client has included valid credentials with its request, check the value of the **IsAuthenticated** property.

Some clients may require that the session be secure if authentication is requested or display warning messages to the user if the connection is not secure. If your application will require clients to authenticate before accessing specific resources, it is recommended that you enable security by setting the **Secure** property to **True** prior to starting the server.

See Also

SendError Method, SendResponse Method, OnAuthenticate Event, OnCommand Event

Reset Method

Reset the internal state of the control to its default values.

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. Is 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

Disconnect Method, Initialize Method, Stop Method, Uninitialize Method

ResolvePath Method

Resolve a path to its full virtual or local file name.

Syntax

object.**ResolvePath**(*ClientId*, *SourcePath*, *ResolvedPath*, [*IsVirtual*])

Parameters

ClientId

An integer that identifies the client session.

SourcePath

A string that specifies the name of the path to resolve. This may either be a virtual path, or a path to a local file name or directory.

ResolvedPath

A string that will contain the resolved path when the method returns.

IsVirtual

An optional Boolean parameter that specifies if the source path is a virtual path or local path.

Return Value

A value of zero is returned if the path could be resolved. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ResolvePath** method is used to resolve a local file name or directory to obtain its virtual path name, or obtain the full path name of a file or directory that is mapped to a virtual path. If the ***IsVirtual*** parameter is omitted or is **False**, the ***SourcePath*** parameter is considered to be a path to a local file or directory and the ***ResolvedPath*** parameter will contain the virtual path. If the ***IsVirtual*** parameter is **True**, then the ***SourcePath*** parameter is considered to be a virtual path and the ***ResolvedPath*** parameter will contain the full path to the local file or directory that the virtual path is mapped to

A virtual path for the client is either relative to the server root directory, or the client home directory if the user was specified in the request URI. These virtual paths are what the client will see as an absolute path on the server. For example, if the server was configured to use "C:\ProgramData\MyServer" as the root directory, and the ***SourcePath*** parameter was specified as "C:\ProgramData\MyServer\Documents\Research", this method would return the virtual path to that directory as "/Documents/Research".

See Also

LocalPath Property, VirtualPath 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 server was restarted, 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. The **OnDisconnect** event will not fire for those client sessions that are terminated when the server is restarted.

See Also

Resume Method, Start Method, Stop Method, Suspend Method

Resume Method

Resume accepting new client connections.

Syntax

object.**Restart**

Parameters

None.

Return Value

A value of zero is returned if the server has resumed accepting new connections, 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 after the **Suspend** method was called.

See Also

Restart Method, Start Method, Stop Method, Suspend Method

SendError Method

Send a result code and message to the client in response to a command.

Syntax

object.**SendError**(*ClientId*, *ErrorCode*, [*Message*])

Parameters

ClientId

An integer that identifies the client session.

ErrorCode

An integer value that specifies the error code that should be sent to the client. This value should correspond to the error result codes defined for HTTP in RFC 2616, which are three-digit values in the range of 400 through 599. The function will fail if an invalid error code is specified.

Message

An optional string value that specifies a message to be sent to the client. If this parameter is omitted is an empty string, a default message associated with the result code will be used.

Return Value

A value of zero is returned if the response was sent to the client. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **SendError** method sends a response to the client indicating that an error has occurred, providing a numeric error code and HTML formatted text which may be displayed to the user. The *Message* parameter should provide a brief description of the error that will be included in the output sent to the client. Note that the message should not contain any special formatting control characters or HTML markup.

This method provides a simplified interface for sending an error response to the client. In some cases, a browser may choose to display its own error message to the user in place of the generic HTML document generated by this method. If you want your application to send a customized HTML document for a specific type of error, you should use the **SendResponse** method.

If you wish to redirect the client to use an alternate URL to access the requested resource, it is recommended that you use the **RedirectRequest** method rather than sending an error response.

See Also

RedirectRequest Method, SendResponse Method, OnCommand Event, OnResult Event

SendResponse Method

Send a result code and message to the client in response to a command.

Syntax

object.**SendResponse**(*ClientId*, *ResultCode*, [*Buffer*], [*Length*])

Parameters

ClientId

An integer that identifies the client session.

ResultCode

An integer value that specifies the command result code to be returned to the client.

Buffer

An optional string or byte array that contains data that should be returned to the client in response to a request. If the server does not wish to send any response data to the client, this parameter can be omitted.

Length

An optional integer value that specifies the number of bytes of data that should be sent to the client. If this parameter is omitted, the number of bytes sent is determined by the length of the string buffer or number of bytes in the byte array provided by the caller. This parameter is ignored if no response data is provided.

Return Value

A value of zero is returned if the response was sent to the client. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **SendResponse** method is used to respond to a command issued by the client from within an **OnCommand** event handler. Command responses are normally handled by the server as a normal part of processing a command and this method is only used if the application has implemented custom commands or wishes to modify the standard responses sent by the server.

Result codes must be three digits (in the range of 100 through 999) and although this method will support the use of non-standard result codes, it is recommended that the client application use the standard codes defined in RFC 2616 whenever possible. The use of non-standard result codes may cause problems with HTTP clients that expect specific result codes in response to a particular command.

If you do not wish to return any data to the client in response to its request (for example, if you want the response to only consist of the headers set using the **SetHeader** method), then you can omit the ***Buffer*** and ***Length*** parameter, and should specify a result code of 204. This tells the client that the request was successful and there is no data included with the response.

This method should only be called once in response to a command sent by the client. If a result code has already been sent in response to a command and this method is called, it will fail and return an error. This is necessary because sending multiple result codes in response to a single command may cause unpredictable behavior by the client.

See Also

OnCommand Event, OnResult Event

SetHeader Method

Create or change the value of a response header for the client session.

Syntax

object.**SetHeader**(*ClientId*, *HeaderName*, *HeaderValue*)

Parameters

ClientId

An integer that identifies the client session.

HeaderName

A string that specifies the name of the header field. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value.

HeaderValue

A string variable that contains the new value of the response header.

Return Value

A Boolean value that specifies if the method was successful.

Remarks

The **SetHeader** method will change the value of a response header for the specified client session, typically within an **OnCommand** event handler. If the ***HeaderName*** value matches an existing header field, its value will be replaced. If the header name is not defined, then a new header will be created with the given value. You should not change the value of most standard response header values unless you are certain of the impact that it would have on the normal operation of the client.

If you wish to define a custom header value that would be included in the response to a client request, you should prefix the header name with "X-" to avoid potential conflicts with the standard response headers. For example, if you wanted to identify a customer, you could create a header field with the name "X-Customer-ID" and set the value to the customer ID number. The client application would receive this custom header value as part of the response to its request for a document.

Refer to Hypertext Transfer Protocol Headers for a list of common request and response headers that are used.

See Also

GetAllHeaders Method, GetHeader Method, OnCommand Event

SetVariable Method

Create or change the value of a CGI environment variable for the specified client.

Syntax

object.**SetVariable**(*ClientId*, *VariableName*, *VariableValue*)

Parameters

ClientId

An integer that identifies the client session.

VariableName

A string that specifies the name of the header field. Header names are not case-sensitive and should not include the colon which acts as a delimiter that separates the header name from its value.

VariableValue

A string variable that contains the new value of the response header.

Return Value

A Boolean value that specifies if the method was successful.

Remarks

The **SetVariable** method will change the value of a environment variable for the specified client session, typically within an **OnCommand** event handler. If the ***VariableName*** value matches an existing variable, its value will be replaced. If the variable is not defined, then a new variable will be created with the given value. The value of an environment variable can be obtained using the **GetVariable** function.

The server will automatically create a number of different environment variables that will be passed to a program or script executed by the server. These variables are defined in RFC 3875 as part of the Common Gateway Interface (CGI) 1.1 specification. The following variables are defined by the server and should not be modified directly by the application:

Variable Name	Description
AUTH_TYPE	The authorization scheme used by the server to authenticate the client session
CONTENT_LENGTH	The length of the request data provided by the client
CONTENT_TYPE	The MIME type that identifies the type of content provided by the client
DOCUMENT_ROOT	The full path to the local document root directory on the server
GATEWAY_INTERFACE	The version of the Common Gateway Interface that is being used by the server
PATH_INFO	The resource or sub-resource that is to be returned by the program or script
PATH_TRANSLATED	The path information mapped to the server root document directory structure
QUERY_STRING	The URL encoded query parameters passed to the program or script
REMOTE_ADDR	The network address of the client sending the request to the server

REMOTE_HOST	The same value as the REMOTE_ADDR variable
REMOTE_USER	The username specified as part of the authentication credentials provided by the client
REQUEST_METHOD	The method used by the client to request the resource
REQUEST_URI	The URI for the script provided by the client
SCRIPT_FILENAME	The full path to the program or script on the server
SCRIPT_NAME	The path to the program or script specified by the client
SERVER_NAME	The hostname or IP address of the server that the client connected to
SERVER_PORT	The port number that the client used to connect to the server
SERVER_PORT_SECURE	This variable has a value of "1" if the client connection to the server is secure
SERVER_PROTOCOL	The version of the server protocol used
SERVER_SOFTWARE	The server identity string which specifies the application name and version

In addition to the environment variables listed, the server will also create variables that are prefixed with "HTTP_" that are set to the value of request headers that are not otherwise defined. For example, the HTTP_USER_AGENT variable will be set to the value of the User-Agent header provided by the client as part of the request.

Note that calling the **SetVariable** method from within the **OnExecute** event handler will have no effect because it occurs after the CGI program or script has completed execution. To create or modify environment variables for the client session, it should be done within an **OnCommand** event handler.

This method will not change the environment block for the server process. Each client session is allocated its own private environment block which is inherited by the CGI program. When the client session terminates, the memory allocated for its environment is released.

See Also

GetHeader Method, OnCommand Event, OnExecute Event

Start Method

Start listening for client connections on the specified IP address and port number.

Syntax

object.**Start**([*ServerAddress*], [*ServerPort*], [*Directory*] [*MaxClients*], [*IdleTime*], [*Options*])

Parameters

ServerAddress

An optional string which specifies the local hostname or IP address address that the server should be bound to. If this parameter is an empty string, then 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. If this parameter is omitted, the control will accept connections on the address specified by the value of the **ServerAddress** property.

ServerPort

An optional integer that specifies the port number the server should use to listen for client connections. If a value of zero is specified, the server will use the standard port number 21 to listen for connections, or port 990 if the server is configured to use implicit TLS. 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. If this parameter is omitted, it defaults to the value specified by the **ServerPort** property.

Directory

An optional string that specifies the path to the root directory for the server. If this parameter is omitted, it defaults to the value specified by the **Directory** property. If this property is not set and no directory is specified, the server will use the current working directory as the root directory.

MaxClients

An optional integer value that specifies the maximum number of clients that may connect to the server. If this parameter is omitted, the value specified by the **MaxClients** property will be used. This value can be adjusted after the server has been created by calling the **Throttle** method.

IdleTime

An optional integer value that specifies the number of seconds a client can be idle before the server terminates the session. If this argument is not specified, the value of the **IdleTime** property will be used. The default idle timeout period is 300 seconds (5 minutes).

Options

An optional integer value that specifies specifies one or more server options. This 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. For a list of options, see Server Option Constants.

Return Value

A value of zero is returned if the server was started, 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.

It is recommended that you always specify an absolute path for the server root directory, either by passing the full pathname as an argument to this method or by setting the **Directory** property. If the path includes environment variables surrounded by percent (%) symbols, they will be automatically expanded.

If you have configured the server to permit clients to upload files, you must ensure that your application has permission to create files in the directory that you specify. A recommended location for the server root directory would be a subdirectory of the %ALLUSERSPROFILE% directory. Using the environment variable ensures that your server will work correctly on different versions of Windows. If the root directory does not exist at the time that the server is started, it will be created.

See Also

Restart Method, Resume Method, Stop Method, Suspend Method, Throttle Method

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 server was stopped, 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.

See Also

Restart Method, Resume Method, Start Method, Suspend Method

Suspend Method

Suspend accepting new client connections.

Syntax

object.**Suspend**

Parameters

None.

Return Value

A value of zero is returned if the server has suspended accepting new connections, 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. All subsequent attempts to connect to the server will be rejected by the server. To resume accepting new client connections, call the **Resume** method. This method will not affect those clients that have already established a connection with the server before the **Suspend** method was called.

See Also

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. If this parameter is omitted, the maximum number of clients allowed will be unchanged. The default value is 100 active 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. If this parameter is omitted, the maximum number of clients per address will be unchanged. The default value is 4 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, Resume Method, Start Method, Suspend 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. 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

Hypertext Transfer Server Control Events

Event	Description
OnAuthenticate	The client has requested authentication with the specified username and password
OnCommand	The client has issued a command to the server
OnConnect	The client established a connection to the server
OnDisconnect	The client has disconnected from the server
OnDownload	The client has downloaded a file from the server
OnError	The client encountered an error when handling a client request
OnExecute	The client has executed an external program on the server
OnIdle	The last client has disconnected from the server
OnResult	The command issued by the client has been processed by the server
OnStart	The server has started listening for connections
OnStop	The server has stopped accepting connections and all client sessions are terminated
OnTimeout	The client has exceeded the maximum allowed idle time
OnUpload	The client has uploaded a file to the server

OnAuthenticate Event

The client has requested authentication with the specified username and password.

Syntax

Sub *object*_**OnAuthenticate** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *HostName* **As Variant**, **ByVal** *UserName* **As Variant**, **ByVal** *Password* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

HostName

A string that specifies the host name that the client used to establish the connection.

UserName

A string that specifies the user name provided by the client.

Password

A string that specifies the password provided by the client.

Remarks

The **OnAuthenticate** event occurs when the client has included authentication credentials with its request. The event handler can call the **Authenticate** method to authenticate the client session. If the client session is not authenticated, the server will send an error response to the client and close the connection.

In most cases, a client will not provide credentials unless the server indicates that they are required to access a specific resource. If you wish to password protect documents in a specific folder, use the **AddPath** method to create a virtual path to the folder and include the **httpAccessProtected** permission. The server will automatically require the client to provide credentials when accessing those documents. To require authentication for a specific resource, implement an **OnCommand** event handler and check the value of the **IsAuthenticated** property when that resource is requested. If the property returns **False**, then use the **RequireAuthentication** method to indicate to the client that it must provide authentication credentials.

If the application has created one or more virtual users using the **AddUser** method and/or the **LocalUser** property has been set to **True**, it is not necessary to implement an **OnAuthenticate** handler unless you also wish to perform custom authentication for specific users.

See Also

AddPath Method, AddUser Method, Authenticate Method, RequireAuthentication Method, OnCommand Event

OnCommand Event

The client has issued a command to the server.

Syntax

Sub *object*_**OnCommand** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *Command* **As Variant**, **ByVal** *Resource* **As Variant**, **ByVal** *Parameters* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

Command

A string that specifies the command that was sent to the server.

Resource

A string that specifies the resource that the client has requested. Depending on the command issued, it may be a document, a folder or an executable script.

Parameters

A string that specifies any query parameters that have been provided by the client. The string will be empty if there were no query parameters included with the request. The query parameters in this string will be URL encoded.

Remarks

The **OnCommand** event occurs after the client has sent a command to the server, but before the command has been processed. This event occurs for all commands issued by the client, including invalid or disabled commands. If the application wishes to handle the command itself, it must perform any processing and then call the **SendResponse** method to return data to the client, or the **SendError** method to send an error response. If no response to the request is sent from within the event handler, then the server will perform its default processing for the command.

After the command has been processed, the **OnResult** event handler will be invoked.

See Also

CommandLine Property, SendError Method, SendResponse Method, OnResult Event

OnConnect Event

The client has established a connection to the server.

Syntax

Sub *object*_**OnConnect** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *ClientAddress* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

ClientAddress

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.

Remarks

The **OnConnect** event occurs after the client has established its initial connection to the server, after the server has checked the active client limits and the TLS handshake has been performed if required. If the server has been suspended, or the limit on the maximum number of client sessions has been exceeded, the server will terminate the client session prior to this event handler being invoked.

If no event handler is implemented, the server will perform the default action of accepting the connection and waiting for the client to send its request. To reject a connection, call the **SendError** method to send an error response to the client. If you do not wish to send an error response, you may simply call the **Disconnect** method to terminate the session.

See Also

OnCommand Event, OnDisconnect Event

OnDisconnect Event

The client has disconnected from the server.

Syntax

Sub *object*_**OnDisconnect** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

Remarks

The **OnDisconnect** event occurs when the client disconnects from the server or when the server terminates the connection to the client by calling the **Disconnect** method. It is not required for the application to explicitly disconnect the client within the event handler, and the application cannot prevent the client from disconnecting from the server.

This event may not occur for a each client session when the server is reset or the control instance is disposed without the application first calling the **Stop** method to shutdown the server.

See Also

OnCommand Event, OnConnect Event

OnDownload Event

The client has successfully downloaded a file from the server.

Syntax

Sub *object*_**OnDownload** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *FileName* **As Variant**, **ByVal** *FileSize* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

FileName

A string that specifies the full path name of the file on the server that was downloaded.

FileSize

An integer value that specifies the number of bytes of data that was downloaded by the client.

Remarks

The **OnDownload** event occurs after the client has successfully downloaded a file from the server using the GET command. If the file transfer fails or is aborted, this event will not occur.

See Also

OnCommand Event, OnUpload Event

OnError Event

The client encountered an error when handling a client request.

Syntax

Sub *object*_**OnError** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *ErrorCode* **As Variant**, **ByVal** *Description* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

ErrorCode

An integer value which specifies the error that has occurred.

Description

A string that describes the error.

Remarks

The **OnError** event occurs whenever the server encounters an error while accepting a client connection or processing a request. It is important to note that this event is not raised for every error that occurs. 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.
- A network error occurs when attempting to process a command issued 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.

See Also

LastError Property, LastErrorString Property, ThrowError Property

OnExecute Event

The client has executed an external program on the server.

Syntax

Sub *object*_**OnExecute** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *Resource* **As Variant**, **ByVal** *Parameters* **As Variant**, **ByVal** *Output* **As Variant**, **ByVal** *ExitCode* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

Resource

A string that specifies the resource that the client has requested.

Parameters

A string that specifies any query parameters that have been provided by the client. The string will be empty if there were no query parameters included with the request. The query parameters in this string will be URL encoded.

Output

A string that contains the standard output of the program that was executed. The format of this output depends on the application that was executed. If the program outputs control characters or other binary data, it will be replaced by spaces to ensure that only printable text is returned.

ExitCode

An integer value that specifies the exit code that was returned by the program.

Remarks

The **OnExecute** event occurs after the client has successfully executed an external CGI program or script.

External programs must be registered by the server application using the **RegisterProgram** method. To enable the use of scripts, the **RegisterHandler** method can be used to associate an executable program with a specific file extension.

See Also

RegisterHandler Method, RegisterProgram Method, OnCommand Event

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

OnResult Event

The command issued by the client has been processed by the server.

Syntax

Sub *object*_**OnResult** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *Resource* **As Variant**, **ByVal** *ResultCode* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

Resource

A string that specifies the resource that was requested by the client.

ResultCode

An integer value that specifies the result code that was sent to the client.

Remarks

The **OnResult** event occurs after the server has processed a command issued by the client. This event will inform the application whether the command that was issued by the client was successful or not. If the command was successful, then other related events such as **OnExecute** may also fire after this event.

The ***ResultCode*** parameter is a three-digit numeric code that is used to indicate success or failure. These codes are defined as part of the Hypertext Transfer Protocol standard, with values in the range of 200-299 indicating success. Values in the range of 400-499 and 500-599 indicate failure due to various error conditions. Examples of such failures would be attempting to access a file that does not exist, issuing an unrecognized command or attempting to perform a privileged operation.

See Also

OnCommand 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 is generated 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 is generated 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 client has exceeded the maximum allowed idle time.

Syntax

Sub *object*_**OnTimeout** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *Elapsed* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

Elapsed

An integer value that specifies the number of seconds that have elapsed.

Remarks

The **OnTimeout** event occurs after the client has has exceeded the maximum allowed idle time, and immediately before the client is disconnected from the server. This event will never occur during a file transfer or directory listing.

To change the default idle timeout period for all clients, set the **IdleTime** property prior to starting the server. To set the idle timeout period for a specific client, set the **ClientIdle** property in an **OnConnect** event handler.

See Also

ClientIdle Property, IdleTime Property, OnConnect Event

OnUpload Event

The client has successfully uploaded a file to the server.

Syntax

Sub *object*_**OnUpload** ([*Index* **As Integer**,] **ByVal** *ClientId* **As Variant**, **ByVal** *FileName* **As Variant**, **ByVal** *FileSize* **As Variant**)

Parameters

ClientId

An integer value which uniquely identifies the client session.

FileName

A string that specifies the full path name of the file on the server that was created or replaced.

FileSize

An integer value that specifies the number of bytes of data that was uploaded by the client.

Remarks

The **OnUpload** event occurs after the client has successfully uploaded a file to the server using the PUT command. If the file transfer fails or is aborted, this event will not occur.

See Also

OnCommand Event, OnUpload Event

Internet Control Message Protocol Control

Determine if a remote host is reachable and how packets of data are routed to that system.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	IcmpClientCtl.IcmpClient
File Name	CSICMX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.IcmpClient.11
ClassID	EFBC543A-466D-4C07-A884-F28590D095BD
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 792

Overview

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

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows

operating system.

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 Control Message Protocol Properties

Property	Description
AutoResolve	Determines if host names and IP addresses are automatically resolved
Blocking	Gets and sets the blocking state of the control
HostAddress	Gets and sets the IP address of the remote host
HostName	Gets and sets the name of the remote host
Interval	Gets and sets the number of milliseconds between ICMP echo packets
IsBlocked	Return if the control is blocked performing an operation
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 to occur
PacketSize	Gets and sets the size of an ICMP echo datagram
RecvCount	Returns the number of packets that have been received
SendCount	Returns the number of packets that have been sent
ThrowError	Enable or disable error handling by the container of the control
Timeout	Gets and sets the amount of time until a blocking operation fails
TimeToLive	Gets and sets the time-to-live value for the ICMP datagram
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
TripAverage	Returns the average packet trip time in milliseconds
TripMaximum	Returns the maximum packet trip time in milliseconds
TripMinimum	Returns the minimum packet trip time in milliseconds
Version	Return the current version of the object

AutoResolve Property

Determines if host names and IP 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 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 **AutoResolve** to False.

Data Type

Boolean

See Also

HostAddress Property, HostName Property

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 read data when none has been written, an error is generated. Events such as **OnEcho** and **OnReply** are only fired if the connection is non-blocking.

Data Type

Boolean

See Also

IsBlocked Property, OnEcho Event, OnReply Event

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, HostName Property

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

Interval Property

Gets and sets the number of milliseconds between ICMP echo packets.

Syntax

object.**Interval** [= *msecs*]

Remarks

The Interval property determines the number of milliseconds the control waits until an ICMP echo packet is sent to the target system. If the interval is set to 0, no more packets are sent.

Data Type

Integer (Int32)

See Also

PacketSize Property, SendCount Property, RecvCount Property, OnEcho Event, OnReply 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.

Note that this property will return True if there is ***any*** blocking operation being performed by the application, regardless if the specified control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

Blocking Property, LastError Property

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, OnError Event

LastErrorString Property

Return a description of the last error to occur

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

PacketSize Property

Gets and sets the size of an ICMP echo datagram.

Syntax

object.**PacketSize** [= *bytes*]

Remarks

The **PacketSize** property determines the size of an ICMP echo packet. The default packet size is 32 bytes. The minimum packet size is 1 byte and the maximum packet size is 65,535 bytes. Specifying a packet size outside of this range will result in an error. Note that packet sizes over 512 bytes may not be supported by your local networking hardware or intermediate routers.

Data Type

Integer (Int32)

See Also

Interval Property, Echo Method

RecvCount Property

Returns the number of packets that have been received.

Syntax

object.**RecvCount**

Remarks

The **RecvCount** property returns the number of packets that have been echoed back from the remote system.

Data Type

Integer (Int32)

See Also

SendCount Property

SendCount Property

Returns the number of packets that have been sent.

Syntax

object.**SendCount**

Remarks

The **SendCount** property returns the number of packets that have been echoed to the remote system.

Data Type

Integer (Int32)

See Also

RecvCount Property

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
IcmpClient1.ThrowError = False
nError = IcmpClient1.Echo(strHostName)

If nError > 0 Then
    MsgBox IcmpClient1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

IcmpClient1.ThrowError = True
IcmpClient1.Echo strHostName

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *milliseconds*]

Remarks

Setting the **Timeout** property specifies the number of milliseconds until a blocking operation fails and the control returns an error.

Data Type

Integer (Int32)

TimeToLive Property

Gets and sets the time-to-live value for the ICMP datagram.

Syntax

object.**TimeToLive** [= *value*]

Remarks

The time-to-live (TTL) value is specified in the IP header of a datagram, and is used to control the number of routers that the datagram is passed through. Each router that handles the datagram decrements the TTL value by one. When it drops to zero, a datagram is returned to the sender, specifying that the TTL has been exceeded.

Setting this property changes the default TTL value for all subsequent ICMP datagrams sent by the control, with the default value being 255. Reading this property returns the value of the TTL field in the IP header of the last datagram received.

Data Type

Integer (Int32)

See Also

PacketSize Property, Timeout Property, Echo Method

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
icmpTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
icmpTraceError	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.
icmpTraceWarning	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.
icmpTraceHexDump	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.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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

TripAverage Property

Returns the average packet trip time in milliseconds.

Syntax

object.**TripAverage**

Remarks

The **TripAverage** property returns the average number of milliseconds for an ICMP echo packet to be returned from the remote host.

Data Type

Integer (Int32)

See Also

Interval Property, RecvCount Property, SendCount Property, TripMaximum Property, TripMinimum Property

TripMaximum Property

Returns the maximum packet trip time in milliseconds.

Syntax

object.**TripMaximum**

Remarks

The **TripMaximum** property returns the maximum number of milliseconds for an ICMP echo packet to be returned from the remote host.

Data Type

Integer (Int32)

See Also

Interval Property, RecvCount Property, SendCount Property, TripAverage Property, TripMinimum Property

TripMinimum Property

Returns the minimum packet trip time in milliseconds.

Syntax

object.**TripMinimum**

Remarks

The **TripMinimum** property returns the minimum number of milliseconds for an ICMP echo packet to be returned from the remote host.

Data Type

Integer (Int32)

See Also

Interval Property, RecvCount Property, SendCount Property, TripAverage Property, TripMaximum Property

Version Property

Return the current version of the object.

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

Internet Control Message Protocol Methods

Method	Description
Cancel	Cancels the current blocking network operation
Echo	Send an ICMP echo datagram to the specified host
Initialize	Initialize the control and validate the runtime license key
Reset	Reset the internal state of the control
TraceRoute	Send a series of ICMP echo datagrams to trace the route taken from the local system to the remote host
Uninitialize	Uninitialize the control and release any system resources that were allocated

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

Echo Method

Send an ICMP echo datagram to the specified host.

Syntax

object.**Echo**([*RemoteHost*], [*Timeout*], [*TimeToLive*])

Parameters

RemoteHost

A string which specifies the host name or IP address which the ICMP echo datagram will be sent to. If this argument is omitted, the value of the **HostAddress** property will be used. If the **HostAddress** property has not been set, then the value of the **HostName** property will be used as the default value.

Timeout

An integer value which specifies the number of milliseconds until a blocking operation fails and the control returns an error. If this argument is omitted, the value of the **Timeout** property will be used as the default value.

TimeToLive

An integer value which specifies the time-to-live (TTL) value for the ICMP echo datagram. If this argument is omitted, the value of the **TimeToLive** property will be used as the default value. For more information about how this is used, refer to the **TimeToLive** property.

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 **Echo** method sends an ICMP echo datagram to the specified host. The failure for a host to respond to an ICMP echo datagram may not indicate a problem with the remote system. In some cases, a router between the local and remote host may be malfunctioning or discarding the datagrams. Systems can also be configured to specifically ignore ICMP echo datagrams and not respond to them; this is often a security measure to prevent certain kinds of Denial of Service attacks.

The ability to send ICMP datagrams may be restricted to users with administrative privileges, depending on the policies and configuration of the local system. If you are unable to send or receive any ICMP datagrams, it is recommended that you check the firewall settings and any third-party security software that could impact the normal operation of this component.

See Also

Blocking Property, HostAddress Property, HostName Property, Timeout Property, TimeToLive Property, TraceRoute Method, OnEcho Event, OnReply Event

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 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 icmpClient = CreateObject("SocketTools.IcmpClient.11")

nError = icmpClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize 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, open network connections will be closed and any handles allocated by the control will be released.

See Also

Cancel Method, Initialize Method, Uninitialize Method

TraceRoute Method

Send a series of ICMP echo datagrams to trace the route taken from the local system to the remote host.

Syntax

object.**TraceRoute**([*RemoteHost*], [*MaxHops*], [*Timeout*])

Parameters

RemoteHost

An optional string which specifies the host name or IP address which the ICMP echo datagram will be sent to. If this argument is omitted, the value of the **HostAddress** property will be used. If the **HostAddress** property has not been set, then the value of the **HostName** property will be used as the default value.

MaxHops

An optional integer value which specifies the maximum number of routers the datagram will be forwarded through (the number of hops) to the remote host. The minimum value is 1 and the maximum value is 255. It is recommended that most applications specify a value of at least 30. If this argument is omitted, the default value of 30 will be used.

Timeout

An optional integer value which specifies the number of milliseconds until a blocking operation fails and the control returns an error. If this argument is omitted, the value of the **Timeout** property will be used as the default value.

Return Value

The method returns the total number of hops from the local system to the remote host. If the method fails, it will return a value of -1. Check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **TraceRoute** method sends a series of ICMP echo datagrams to the specified host, adjusting the time-to-live value to determine the intermediate hosts that route the packet. This method will always block until the trace completes, regardless of how the **Blocking** property is set. The **OnTrace** event will fire for each intermediate host along the route.

It is important to note that the failure of an intermediate host to respond to an ICMP echo datagram may not indicate a problem with the remote system. Systems can be configured to specifically ignore ICMP echo datagrams and not respond to them; this is often a security measure to prevent certain kinds of Denial of Service attacks.

The ability to send ICMP datagrams may be restricted to users with administrative privileges, depending on the policies and configuration of the local system. If you are unable to send or receive any ICMP datagrams, it is recommended that you check the firewall settings and any third-party security software that could impact the normal operation of this component.

See Also

HostAddress Property, HostName Property, Timeout Property, Echo Method, OnTrace 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

Internet Control Message Protocol Events

Event	Description
OnCancel	This event is generated when a blocking operation is canceled
OnEcho	The OnEcho event is generated when a packet is sent to the remote host
OnError	This event is generated when a control error occurs
OnReply	The OnReply event is generated when reply to the ICMP echo datagram is received
OnTimeout	This event is generated when a blocking operation times out
OnTrace	This event is generated for each intermediate host when tracing the route from the local system to a remote host

OnCancel Event

The **OnCancel** event is generated when a blocking operation is canceled.

Syntax

Sub *object*_**OnCancel** ([*Index* **As Integer**])

Remarks

This event is generated when a blocking operation on the socket, such as sending or receiving data, is canceled with the **Cancel** method.

See Also

Cancel Method, OnError Event, OnTimeout Event

OnEcho Event

The OnEcho event is generated when a packet is sent to the remote host.

Syntax

Sub *object*_**OnEcho**([*Index* **As Integer**,] **ByVal** *SequenceId* **As Variant**, **ByVal** *RemoteHost* **As Variant**, **ByVal** *PacketSize* **As Variant**)

Remarks

The **OnEcho** event is generated for non-blocking sockets when an ICMP echo datagram is sent to the remote host. This event is only generated when the **Blocking** property is set to False. The following arguments are passed to the event handler:

SequenceId

An integer which specifies the packet sequence number, which is used to uniquely identify each packet that is sent to the remote host. This value will increase for each ICMP echo datagram that is sent until the remote host address is changed. Once a new remote host is specified, the sequence number is reset.

RemoteHost

A string which specifies the host name or IP address that the echo datagram was sent to.

PacketSize

An integer which specifies the size of the ICMP echo datagram that was sent to the remote host.

See Also

Blocking Property, Echo Method, OnReply 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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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

OnReply Event

The OnReply event is generated when reply to the ICMP echo datagram is received.

Syntax

Sub *object*_**OnReply**([*Index* **As Integer**,] **ByVal** *SequenceId* **As Variant**, **ByVal** *RemoteHost* **As Variant**, **ByVal** *PacketSize* **As Variant**, **ByVal** *ElapsedTime* **As Variant**)

Remarks

The **OnReply** event is fired when a packet is echoed back from the remote system. Note that there is no guarantee that packets will be returned in the same sequence order they were sent or that they will be returned at all. This event is only generated when the **Blocking** property is set to False. The following arguments are passed to the event handler:

SequenceId

An integer which specifies the packet sequence number, which is used to uniquely identify each packet that is sent to the remote host. This value will increase for each ICMP echo datagram that is received.

RemoteHost

A string which specifies the host name or IP address that returned the echo datagram.

PacketSize

An integer which specifies the size of the ICMP echo datagram that was received.

ElapsedTime

An integer which specifies the number of milliseconds that have elapsed since the packet was sent by the control. This value can be used to measure the latency between the local system and remote host.

See Also

Blocking Property, Echo Method, OnEcho 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. This event is only triggered when the **Blocking** property is set to True.

See Also

Blocking Property, Timeout Property, OnCancel Event

OnTrace Event

The **OnTrace** event is generated for each intermediate host when tracing the route from the local system to a remote host.

Syntax

Sub *object*_**OnTrace**([*Index* **As Integer**,] **ByVal** *Distance* **As Variant**, **ByVal** *RemoteHost* **As Variant**, **ByVal** *TripAverage* **As Variant ByVal** *TripMaximum* **As Variant ByVal** *TripMinimum* **As Variant**)

Remarks

The **OnTrace** event is generated when the **TraceRoute** method is called. This event will fire for each intermediate host in the route from the local system and the remote host. The following arguments are passed to the event handler:

Distance

An integer which specifies distance from the local system to the specified host. This value represents the number of times that the packet was forwarded through a router, also known as the number of "hops" to the remote host. With a trace route, this value will start at one and increment by one for each intermediate host until the destination is reached or the operation times out.

RemoteHost

A string which specifies the host name or IP address for the host along the route. If the **AutoResolve** property is True, then the control will attempt to determine the host name of the system. If the host name can not be determined, or the property is False, then this argument will specify the IP address. Note that performing the reverse DNS lookup to resolve the host name can be time consuming.

TripAverage

An integer which specifies the average number of milliseconds that it took for a series of ICMP echo packets to be returned from the host.

TripMaximum

An integer which specifies the maximum number of milliseconds that it took for a series of ICMP echo packets to be returned from the host.

TripMinimum

An integer which specifies the minimum number of milliseconds that it took for a series of ICMP echo packets to be returned from the host.

See Also

AutoResolve Property, TraceRoute Method

Internet Message Access Protocol Control

Manage email messages and mailboxes on a mail server.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	ImapClientCtl.ImapClient
File Name	CSMAPX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.ImapClient.11
ClassID	63472DEA-CB51-4A3F-8886-703D7AC887E2
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 3501

Overview

The Internet Message Access Protocol (IMAP) is an application protocol which is used to access a user's email messages which are stored on a mail server. However, unlike the Post Office Protocol (POP) where messages are downloaded and processed on the local system, the messages on an IMAP server are retained on the server and processed remotely. This is ideal for users who need access to a centralized store of messages or have limited bandwidth. For example, traveling salesmen who have notebook computers or mobile users on a wireless network would be ideal candidates for using IMAP.

The SocketTools IMAP control implements the current standard for this protocol, and provides functions to retrieve messages, or just certain parts of a message, create and manage mailboxes, search for specific messages based on certain criteria and so on. The interface is designed as a superset of the Post Office Protocol interface, so developers who are used to working with the POP3 control will find the IMAP control very easy to integrate into an existing application.

This control supports secure connections using the TLS protocol.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical

updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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 Message Access Protocol Control Properties

Property	Description
AuthType	Gets and sets the method used to authenticate the user
AutoResolve	Determines if host names and IP addresses are automatically resolved
BearerToken	Gets and sets the OAuth 2.0 bearer token used for authentication
Blocking	Gets and sets the blocking state of the control
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 client certificate
CertificatePassword	Gets and sets the password associated with the client certificate
CertificateStatus	Return the status of the server certificate
CertificateStore	Gets and sets the name of the client 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
Delimiter	Returns the hierarchical path delimiter used for the current mailbox
HashStrength	Return the length of the message digest that was selected
HeaderField	Gets and sets the current header field name
HeaderValue	Return the value of the current header field
HostAddress	Gets and sets the IP address of the mail server
HostName	Gets and sets the host name of the mail server
IsBlocked	Return if the control is blocked performing an operation
IsConnected	Determine if the control is connected to a server
IsInitialized	Determine if the control has been initialized
IsReadable	Return if data can be read from the server without blocking
IsWritable	Return if data can be sent to the server without blocking
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error to occur
Mailbox	Returns the name of the specified mailbox from a list of mailboxes on the server
Mailboxes	Returns the number of mailboxes available on the server
MailboxFlags	Returns one or more flags which identify characteristics of the current mailbox
MailboxMask	Gets and sets the current mailbox wildcard mask
MailboxName	Gets and sets the name of the current mailbox

MailboxPath	Gets and sets the current mailbox reference path
MailboxSize	Return the size of the current mailbox
MailboxUID	Returns the unique identifier for the current mailbox
Message	Gets and sets the current message number
MessageCount	Return the number of messages available in the current mailbox
MessageFlags	Returns one or more flags which identify characteristics of the current message
MessagePart	Return the contents of the specified part in a multipart message
MessageParts	Return the number of parts in the current message
MessageSize	Return the size of the current message in bytes
MessageUID	Return the UID for the current message on the mail server
NewMessages	Return the number of new messages available in the current mailbox
Options	Gets and sets the options that are used in establishing a connection
Password	Gets and sets the password for the current user
ReadOnly	Determine if the mailbox can be modified
RecentMessages	Returns the number of messages which have recently arrived in the mailbox
RemotePort	Gets and sets the port number for a remote connection
ResultCode	Return the result code of the previous action
ResultString	Return a string describing the results of the previous action
Secure	Set or return if a connection to the server is secure
SecureCipher	Return the encryption algorithm used to establish the secure connection with the server
SecureHash	Return the message digest selected when establishing the secure connection with the server
SecureKeyExchange	Return the key exchange algorithm used to establish the secure connection with the server
SecureProtocol	Gets and sets the security protocol used to establish the secure connection with the server
Subscribed	Determines if the user has subscribed to the currently selected mailbox
ThrowError	Enable or disable error handling by the container of the control
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
UnreadMessages	Returns the number of unread messages in the current mailbox
UserName	Gets and sets the current user name
Version	Return the current version of the object

AuthType Property

Gets and sets the method used to authenticate the user.

Syntax

object.**AuthType** [= *type*]

Remarks

The **AuthType** property specifies the type of authentication that should be used when the client connects to the mail server. The following authentication methods are supported:

Value	Description
imapAuthLogin	This authentication type will use the LOGIN method to authenticate the client session. This encodes the username and password in a specific format, but the credentials are not encrypted. It should be used over a secure connection. The server must support the Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4422.
imapAuthPlain	This authentication type will use the PLAIN method to authenticate the client session. This encodes the username and password in a specific format, but the credentials are not encrypted. It should be used over a secure connection. The server must support the PLAIN Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4616.
imapAuthXOAuth2	This authentication type will use the XOAUTH2 method to authenticate the client session. This authentication method does not require the user password, instead the **BearerToken** property must specify the OAuth 2.0 bearer token issued by the service provider.
imapAuthBearer	This authentication type will use the OAUTHBEARER method to authenticate the client session as defined in RFC 7628. This authentication method does not require the user password, instead the **BearerToken** property must specify the OAuth 2.0 access token issued by the service provider.

Data Type

Integer (Int32)

Remarks

The **imapAuthXOAuth2** and **imapAuthBearer** authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Changing the value of the **BearerToken** property will automatically set the current authentication method to use OAuth 2.0.

See Also

BearerToken Property, Password Property, UserName Property, Connect Method

AutoResolve Property

Determines if host names and IP 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 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 **AutoResolve** to False.

Data Type

Boolean

See Also

HostAddress Property, HostName Property

BearerToken Property

Gets and sets the OAuth 2.0 bearer token for the current user.

Syntax

object.**BearerToken** [= *token*]

Remarks

The **BearerToken** property specifies the OAuth 2.0 bearer token used to authenticate the user. This property is used as the default value for the **Connect** method if the token is not provided as an parameter.

Assigning a value to this property will change the current authentication method to use OAuth 2.0 if necessary.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an bearer token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

If the current authentication method does not use OAuth 2.0, this property will return an empty string and you should check the value of the **Password** property to obtain the current user's password. Refer to the **AuthType** property for more information on the available authentication methods.

Data Type

String

See Also

AuthType Property, Password Property, UserName Property, Connect 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 read data when none has been written, an error is generated. Events such as **OnConnect**, **OnDisconnect**, **OnRead** and **OnWrite** are only fired if the connection is non-blocking.

Data Type

Boolean

See Also

IsBlocked Property, IsReadable Property, IsWritable Property

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 = ImapClient1.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 client certificate.

Syntax

object.**CertificateName** [= *name*]

Remarks

This property sets the common name or friendly name of the certificate that should be used to establish the connection with the server. 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. If a certificate name is specified, the certificate must have a private key associated with it, otherwise the connection attempt 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 client 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
stCertificateNone	No certificate information is available. A secure connection was not established with the server.
stCertificateValid	The certificate is valid.
stCertificateNoMatch	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.
stCertificateExpired	The certificate has expired and is no longer valid. The client can examine the **CertificateExpires** property to determine when the certificate expired.
stCertificateRevoked	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.
stCertificateUntrusted	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.
stCertificateInvalid	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:

```
'
' Initialize the control properties
```

```
'

ImapClient1.HostName = strHostName
ImapClient1.Secure = True

nError = ImapClient1.Connect()
If nError > 0 Then
     MsgBox "Unable to connect to server " & strHostName, vbExclamation
     Exit Sub
End If

If ImapClient1.CertificateStatus <> stCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          ImapClient1.Disconnect
          Exit Sub
     End If
End If

ImapClient1.Disconnect
```

See Also

CertificateExpires Property, CertificateIssued Property, CertificateIssuer Property, CertificateSubject Property, Secure Property

CertificateStore Property

Gets and sets the name of the client certificate store or file.

Syntax

object.**CertificateStore** [= *store*]

Remarks

This property sets the name of the certificate store that contains the client certificate that should be used when establishing a secure connection with 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 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

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 = ImapClient1.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 client certificate.

Syntax

object.**CertificateUser** [= *username*]

Remarks

This property sets the name of the user that owns the client certificate that will be used to establish a secure connection with the server. 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

Delimiter Property

Returns the hierarchical path delimiter used for the current mailbox.

Syntax

object.**Delimiter**

Remarks

The **Delimiter** property returns a string which specifies the path delimiter used for the current mailbox. If the IMAP server supports multiple levels of mailboxes, then a special character or sequence of characters are used as delimiters between different levels of the mailbox hierarchy. On most systems, including most UNIX and Windows platforms, the delimiter is the forward slash "/" character.

It is possible that an IMAP server may only support a flat namespace, in which case this property will return an empty string.

Data Type

String

See Also

MailboxName Property, CreateMailbox Method, DeleteMailbox Method, SelectMailbox 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

HeaderField Property

Gets and sets the current header field name.

Syntax

object.**HeaderField** [= *header*]

Remarks

The **HeaderField** property returns the name of the current header field. Setting this property causes the control to determine if that header field exists, and if it does, to update the **HeaderValue** property with its value. This property can be used to obtain the value of any header in the current message.

Data Type

String

See Also

HeaderValue Property, Message Property, MessageUID Property

HeaderValue Property

Return the value of the current header field.

Syntax

object.**HeaderValue**

Remarks

The **HeaderValue** property returns the value of the header field specified by the **HeaderField** property. This property can be used to obtain the value of any header in the current message.

Data Type

String

See Also

HeaderField Property, Message Property, MessageUID Property

HostAddress Property

Gets and sets the IP address of the server.

Syntax

object.**HostAddress** [= *ipaddress*]

Remarks

The **HostAddress** property can be used to set the IP address for a server 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, HostName Property

HostName Property

Gets and sets the name of the server.

Syntax

object.**HostName** [= *hostname*]

Remarks

The **HostName** property should be set to the name of the server 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

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.

Note that this property will return True if there is ***any*** blocking operation being performed by the application, regardless if the specified control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

Blocking Property, LastError Property

IsConnected Property

Determine if the control is connected to a server.

Syntax

object.**IsConnected**

Remarks

The **IsConnected** read-only property is set to a value of true if the control is connected with a server, otherwise the property has a value of false.

Data Type

Boolean

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

IsReadable Property

Return if data can be read from the server without blocking.

Syntax

object.**IsReadable**

Remarks

The **IsReadable** property returns True if data can be read from the server without blocking. For non-blocking connections, this property can be checked before the application attempts to read the data, preventing an error.

Data Type

Boolean

See Also

IsConnected Property, Read Method, OnRead Event

IsWritable Property

Return if data can be sent to the server without blocking.

Syntax

object.**IsWritable**

Remarks

The **IsWritable** property returns True if data can be written without blocking. For non-blocking connections, this property can be checked before the application attempts to send data to the server, 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 send data without an error. The next operation may result in an **stErrorOperationWouldBlock** or **stErrorOperationInProgress** error. The application must treat these errors as recoverable, and should be prepared to retry operations that result in them.

Data Type

Boolean

See Also

IsReadable Property, Write Method, OnWrite Event

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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

Mailbox Property

Returns the name of the specified mailbox from a list of mailboxes on the server.

Syntax

object.**Mailbox**(*Index*)

Remarks

The **Mailbox** property array is used to enumerate the available mailboxes on the IMAP server. This is a zero-based array, which means that the index value for the first mailbox is zero. The total number of mailboxes that are available on the server is returned by the **Mailboxes** property.

Data Type

String

Example

The following example demonstrates how to use the **Mailbox** property array to populate a listbox that contains the names of the available mailboxes:

```
For nIndex = 0 To ImapClient1.Mailboxes - 1
  List1.AddItem ImapClient1.Mailbox(nIndex)
Next
```

See Also

Mailboxes Property, MailboxFlags Property, MailboxName Property, MailboxUID Property, ReadOnly Property

Mailboxes Property

Returns the number of mailboxes available on the server.

Syntax

object.**Mailboxes**

Remarks

The **Mailboxes** property returns the total number of mailboxes available to the current account on the server. This property can be used in conjunction with the **Mailbox** property array to enumerate the names of all of the mailboxes which can be selected by the client.

Data Type

Integer (Int32)

Example

The following example demonstrates how to use the **Mailboxes** property to populate a listbox that contains the names of the available mailboxes:

```
For nIndex = 0 To ImapClient1.Mailboxes - 1
  List1.AddItem ImapClient1.Mailbox(nIndex)
Next
```

See Also

Mailbox Property, MailboxFlags Property, MailboxName Property, MailboxUID Property, ReadOnly Property

MailboxFlags Property

Returns one or more flags which identify characteristics of the current mailbox.

Syntax

object.**MailboxFlags**

Remarks

The **MailboxFlags** property returns information about the currently selected mailbox. The value returned is one or more of the following bit flags:

Value	Description
imapFlagNoInferiors	The mailbox does not contain any child mailboxes. In the IMAP protocol, these are referred to as inferior hierarchical mailbox names.
imapFlagMarked	The mailbox is marked as being of interest to a client. If this flag is used, it typically means that the mailbox contains messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.
imapFlagUnmarked	The mailbox is marked as not being of interest to a client. If this flag is used, it typically means that the mailbox does not contain any messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.

Data Type

Integer (Int32)

Example

The following example demonstrates how to check the **MailboxFlags** property to see if the mailbox contains any child mailboxes:

```
If (ImapClient1.MailboxFlags And imapFlagNoInferiors) <> 0 Then
    MsgBox "This mailbox does not contain any child mailboxes"
End If
```

See Also

Mailbox Property, Mailboxes Property, MailboxName Property, MailboxUID Property, ReadOnly Property

MailboxMask Property

Gets and sets the current mailbox wildcard mask.

Syntax

object.**MailboxMask** [= *mask*]

Remarks

The **MailboxMask** property returns the current mailbox wildcard mask. If no wildcard mask has been specified by the client, this property will return an empty string.

Setting the **MailboxMask** property will determine which mailboxes are returned by the **Mailbox** property array. Wildcards may include the asterisk (which matches any mailbox as well as any child mailboxes) and the percent sign (which matches any mailbox, but does not match any child mailboxes). This property may be used in conjunction with the **MailboxPath** property to further qualify which mailboxes are returned.

Data Type

String

See Also

Mailbox Property, Mailboxes Property, MailboxPath Property, SelectMailbox Method, UnselectMailbox Method

MailboxName Property

Gets and sets the name of the current mailbox.

Syntax

object.**MailboxName** [= *mailbox*]

Remarks

The **MailboxName** property returns the name of the currently selected mailbox. If no mailbox has been selected by the client, this property will return an empty string.

Setting the **MailboxName** property will select a new mailbox in read-write mode. If the client has a different mailbox currently selected, that mailbox will be closed and any messages marked for deletion will be expunged. To prevent deleted messages from being removed from the previous mailbox, call the **UnselectMailbox** method prior to selecting the new mailbox. Setting the **MailboxName** property to an empty string will cause the current mailbox to be unselected, and a new mailbox will not be selected. Before the application can access any messages, it must select a new mailbox.

Selecting a new mailbox will automatically update those properties which provide information about the current mailbox, such as the **MailboxFlags** and **MailboxUID** properties. If an application wishes to update the information for the current mailbox, simply set the **MailboxName** property again with the same mailbox name. Note that this will not cause any messages marked for deletion to be expunged.

The special case-insensitive mailbox name INBOX is used for new messages. Other mailbox names may or may not be case-sensitive depending on the IMAP server's operating system and implementation.

If the mailbox name contains international characters then it is automatically encoded using a modified version of UTF-7 encoding. For example, if a mailbox is named "Håndskrift", the mailbox name created on the server will be the string "H&AOU-ndskrift". The control will automatically decode UTF-7 encoded mailbox names, making the conversion transparent to the application.

Data Type

String

See Also

Mailbox Property, Mailboxes Property, MailboxFlags Property, MailboxUID Property, ReadOnly Property, SelectMailbox Method, UnselectMailbox Method

MailboxPath Property

Gets and sets the current mailbox reference path.

Syntax

object.**MailboxPath** [= *path*]

Remarks

The **MailboxPath** property returns the current mailbox reference path. If no path has been specified by the client, this property will return an empty string.

Setting the **MailboxPath** property will determine which mailboxes are returned by the **Mailbox** property array. Typically this is used to specify a subdirectory where mail folders are stored for the current user. Note that some mail servers may not permit absolute reference paths, and in most cases the path should include a trailing slash. This property may be used in conjunction with the **MailboxMask** property to further qualify which mailboxes are returned.

Data Type

String

See Also

Mailbox Property, Mailboxes Property, MailboxMask Property, SelectMailbox Method, UnselectMailbox Method

MailboxSize Property

Return the size of the current mailbox.

Syntax

object.**MailboxSize**

Remarks

The **MailboxSize** property returns the combined size of all messages in the current mailbox. Reading this property may require a significant amount of time to calculate the mailbox size if there are a large number of messages in the mailbox. Because it can potentially result in long delays, it is not recommended that an application calculate the mailbox size unless it is absolutely necessary.

Data Type

Integer (Int32)

See Also

Message Property, MessageCount Property, MessageSize Property

MailboxUID Property

Returns the unique identifier for the current mailbox.

Syntax

object.**MailboxUID**

Remarks

The **MailboxUID** property returns an integer value which uniquely identifies the mailbox and corresponds to the UIDVALIDITY value returned by the IMAP server. The actual value is determined by the server and should be considered opaque. The protocol specification requires that a mailbox's UID must not change unless the mailbox contents are modified or existing messages in the mailbox have been assigned new UIDs.

An application can use the **MailboxUID** property value in combination with the **MessageUID** property in order to uniquely identify a message on the server. However, the application must take into consideration that the IMAP server can reassign new message UIDs when the mailbox is modified. If the mailbox and message UIDs are being stored on the local system to track what messages have been retrieved from the server, the application must check the UID of the mailbox whenever it is selected. If the mailbox UID has changed, this means that the UIDs for the messages in that mailbox may have changed. The client should resynchronize with the server, and update it's local copy of that mailbox.

Data Type

Integer (Int32)

See Also

Mailbox Property, Mailboxes Property, MailboxFlags Property, MailboxName Property, ReadOnly Property

Message Property

Gets and sets the current message number.

Syntax

object.**Message** [= *value*]

Remarks

The **Message** property sets or returns the message number for the currently selected mailbox. Message numbers range from 1 through the number of messages available on the server, as returned by the **MessageCount** property. Setting the **Message** property to an invalid message number will generate an error.

Data Type

Integer (Int32)

See Also

MessageCount Property, MessageSize Property

MessageCount Property

Return the number of messages available in the current mailbox.

Syntax

object.**MessageCount**

Remarks

The **MessageCount** property returns the number of messages available to be retrieved from the currently selected mailbox.

Data Type

Integer (Int32)

See Also

Message Property, MessageFlags Property, MessagePart Property, MessageParts Property, MessageSize Property, MessageUID Property, GetMessage Method

MessageFlags Property

Returns one or more flags which identify characteristics of the current message.

Syntax

object.**MessageFlags** [= *flags*]

Remarks

The **MessageFlags** property returns information about the currently selected message specified by the **Message** property. The value returned is one or more of the following bit flags:

Value	Value	Description
imapFlagNone	No flags have been set for the current message	
imapFlagAnswered	The message has been answered	
imapFlagDraft	The message is a draft copy and has not been delivered	
imapFlagUrgent	The message has been flagged for urgent or special attention	
imapFlagSeen	The message has been read	
imapFlagRecent	The message has been added to the mailbox recently	
imapFlagDeleted	The message has been marked for deletion	

Setting the **MessageFlags** property changes the flags for the currently selected message. Multiple bit flags can be combined using the bitwise Or operator. An application can test if a flag is set by using the bitwise And operator.

Data Type

Integer (Int32)

Example

The following example demonstrates how to check the **MessageFlags** property to see if the message has been marked for deletion, and if it has, to clear the flag so that it will not be deleted when the mailbox is unselected:

```
If (ImapClient1.MessageFlags And imapFlagDeleted) <> 0 Then
    ImapClient1.MessageFlags = ImapClient1.MessageFlags And Not imapFlagDeleted
End If
```

See Also

Message Property, MessageCount Property, MessagePart Property, MessageParts Property, MessageSize Property, MessageUID Property, GetMessage Method

MessagePart Property

Return the contents of the specified part in a multipart message.

Syntax

object.**MessagePart**(*PartNumber*)

Remarks

The **MessagePart** property returns the contents of the specified message part. All messages have at least one part, which consists of one or more header fields, followed by the body of the message. The default part, part 1, refers to the main message header and body. If the message contains multiple parts (as with a message that contains one or more attached files), the **MessagePart** property can be set to refer to that specific part of the message.

Messages with file attachments typically consist of a message part which describes the contents of the attachment, followed by the attachment itself. For a message with one attached file, there would be a total of three parts. Part 1 would refer to the main message part, which contains the headers such as From, To, Subject, Date and so on. For multipart messages, part 1 typically does not have a message body, since any text is usually created as a separate part (for those messages that do not contain multiple parts, the part 1 body contains the text message). Part 2 would contain the text describing the attachment, and part 3 would contain the attachment itself. If the attached file is binary, then the transfer encoding type would usually be base64.

It is important to note that an IMAP server considers the first part of a multipart message to be part 1, so the **MessagePart** property array is one-based. This is different than the SocketTools MIME control, which considers the first part of a multipart message to be zero.

Data Type

Integer (Int32)

See Also

Message Property, MessageParts Property, GetMessage Method

MessageParts Property

Return the number of parts in the current message.

Syntax

object.**MessageParts**

Remarks

The **MessageParts** property returns the number of parts in the current message. All messages have at least one part, referenced as part 1. Multipart messages will consist of additional parts which may be accessed by reading the **MessagePart** property array or calling the **GetMessage** method.

Data Type

Integer (Int32)

See Also

Message Property, MessagePart Property, GetMessage Method

MessageSize Property

Return the size of the current message in bytes.

Syntax

object.**MessageSize**

Remarks

The **MessageSize** property returns the size of the current message in bytes. The size includes the header and body portion of the message.

Data Type

Integer (Int32)

See Also

Message Property, MessageCount Property

MessageUID Property

Return the UID for the current message on the mail server.

Syntax

object.**MessageUID**

Remarks

The **MessageUID** property returns an integer value which specifies a unique identifier for this message. The actual value is determined by the server and should be considered opaque. If the client application stores the message UID on the local system, it should also store the UID for the mailbox that contains the message. If the mailbox UID changes, the message UID may no longer be valid.

An application can use the **MessageUID** property value in combination with the **MailboxUID** property in order to uniquely identify a message on the server. However, the application must take into consideration that the IMAP server can reassign new message UIDs when the mailbox is modified. If the mailbox and message UIDs are being stored on the local system to track what messages have been retrieved from the server, the application must check the UID of the mailbox whenever it is selected. If the mailbox UID has changed, this means that the UIDs for the messages in that mailbox may have changed. The client should resynchronize with the server, and update it's local copy of that mailbox.

Data Type

Integer (Int32)

See Also

MailboxUID Property, Message Property, MessageFlags Property

NewMessages Property

Return the number of new messages available in the current mailbox.

Syntax

object.**NewMessages**

Remarks

The **NewMessages** property returns the number of new, unread messages available to be retrieved from the currently selected mailbox. To determine the total number of unread messages in the mailbox, regardless of when they were added to the mailbox, use the **UnreadMessages** property.

Data Type

Integer (Int32)

See Also

MessageCount Property, RecentMessages Property, UnreadMessages Property, CheckMailbox Method

Options Property

Gets and sets the options that are used in establishing a connection.

Syntax

object.**Options** [= *value*]

Remarks

The **Options** property is an integer value which specifies one or more options. The value specified for this property will be used as the default options when connecting to the server. The property value is created by using a bitwise operator with one or more of the following values:

Value	Description
imapOptionNone	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used.
imapOptionIdentify	This option specifies the client should identify itself to the server. If enabled, the client will send the ID command to the server as defined in RFC 2971. This option has no effect if the server does not support the ID command.
&H400	imapOptionTunnel	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
&H800	imapOptionTrustedSite	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 the TLS protocol.
&H1000	imapOptionSecureExplicit	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. This option initiates the secure session using the STARTTLS command.

&H2000	imapOptionSecureImplicit	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol, and the secure session must be negotiated immediately after the connection has been established.
&H8000	imapOptionSecureFallback	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.
&H40000	imapOptionPreferIPv6	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.

Data Type

Integer (Int32)

See Also

Secure Property, Connect Method

Password Property

Gets and sets the password for the current user.

Syntax

object.**Password** [= *password*]

Remarks

The **Password** property specifies the password used to authenticate the user. This property is used as the default value for the **Connect** method if no password is specified as an argument.

Refer to the **AuthType** property for more information on the available authentication methods. If you are using the OAuth 2.0 authentication method, this property should not be set to the user's password. Instead, you should set the **BearerToken** property to the OAuth 2.0 bearer token issued by the mail service provider. Note that these access tokens can be much larger than your typical password and are only valid for a limited period of time.

You can use the **Password** property to specify an OAuth 2.0 bearer token. However, it is recommended that you use the **BearerToken** property instead of assigning it to this property. It will ensure compatibility with future versions of the control and make it clear in your code you are using an OAuth 2.0 bearer token and not a password. If the **AuthType** property specifies one of the OAuth 2.0 authentication methods, this property will return the bearer token.

Data Type

String

See Also

AuthType Property, BearerToken Property, UserName Property, Connect Method

ReadOnly Property

Determine if the mailbox can be modified.

Syntax

object.**ReadOnly**

Remarks

The **ReadOnly** property returns True if the currently selected mailbox cannot be modified by the client. A value of false means that the mailbox can be modified.

Data Type

Boolean

See Also

Mailbox Property, Mailboxes Property, MailboxFlags Property, MailboxName Property, MailboxUID Property

RecentMessages Property

Returns the number of messages which have recently arrived in the mailbox.

Syntax

object.**RecentMessages**

Remarks

The **RecentMessages** property returns the number of messages which have been recently added to the currently selected mailbox. This property is particularly useful when the INBOX mailbox is selected because it enables the application to check if any new messages have arrived. To determine the total number of unread messages in the mailbox, use the **UnreadMessages** property.

Data Type

Integer (Int32)

See Also

MessageCount Property, NewMessages Property, UnreadMessages Property, CheckMailbox Method

RemotePort Property

Gets and sets the port number for a remote connection.

Syntax

object.**RemotePort** [= *portnumber*]

Remarks

The **RemotePort** property is used to set the port number that the control will use to establish a connection with the server.

Data Type

Integer (Int32)

See Also

HostAddress Property, HostName Property

ResultCode Property

Return the result code of the previous action.

Syntax

object.**ResultCode**

Remarks

The **ResultCode** read-only property returns the result code of the last action performed by the client. This property should be checked after the **Command** method is used to execute a command on the server to determine if the operation was successful. One of the following result codes may be returned:

Value	Description
imapResultUnknown	An unknown result code was returned by the server.
imapResultOk	The previous command completed successfully. The result string contains information about the results of the command.
imapResultNo	The previous command could not be completed. The result string contains information about why the command failed.
imapResultBad	The previous command could not be completed, the command may be invalid or not supported on the server. The result string contains information about why the command failed.
imapResultContinue	The command has executed and is waiting for additional data from the client.

Data Type

Integer (Int32)

See Also

ResultString Property, Command Method, OnCommand Event

ResultString Property

Return a string describing the results of the previous action.

Syntax

object.**ResultString**

Remarks

The **ResultString** read-only property returns the result string from the last action taken by the client. This string is generated by the server, and typically is used to describe the result code. For example, if an error is indicated by the result code, the result string may describe the condition that caused the error.

Data Type

String

See Also

ResultCode Property, Command Method, OnCommand Event

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 must 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 strongly recommended that any application that sets this property True use error handling to trap an errors that may occur. If the control 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.

Data Type

Boolean

Example

The following example establishes a secure connection to a server:

```
ImapClient1.HostName = strHostName
ImapClient1.Secure = True

nError = ImapClient1.Connect()
If nError > 0 Then
    MsgBox "Unable to connect to server " & strHostName, vbExclamation
    Exit Sub
End If

If ImapClient1.CertificateStatus <> stCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          ImapClient1.Disconnect
          Exit Sub
     End If
End If
```

See Also

CertificateStatus 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
stCipherNone	No cipher has been selected. This is not a secure connection with the server.
stCipherRC2	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.
stCipherRC4	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.
stCipherRC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
stCipherDES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher using 56-bit keys.
stCipherDES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively using a 168-bit key length.
stCipherDESX	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.
stCipherAES	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.
stCipherSkipjack	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
stCipherBlowfish	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
stHashMD5	The MD5 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA1	The SHA-1 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA256	The SHA-256 algorithm has been selected.
stHashSHA384	The SHA-384 algorithm has been selected.
stHashSHA512	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
stKeyExchangeNone	No key exchange algorithm has been selected. This is not a secure connection with the server.
stKeyExchangeRSA	The RSA public key exchange algorithm has been selected.
stKeyExchangeKEA	The KEA public key exchange algorithm has been selected. This is an improved version of the Diffie-Hellman public key algorithm.
stKeyExchangeDH	The Diffie-Hellman public key exchange algorithm has been selected.
stKeyExchangeECDH	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

Subscribed Property

Determines if the user has subscribed to the currently selected mailbox.

Syntax

object.**Subscribed** [= { True | False }]

Remarks

The **Subscribed** property is used to determine if the current mailbox has been subscribed to by the user. If the property returns False, the server has indicated that the user has not subscribed to the mailbox. If the property returns True, the current mailbox is in the user's subscription list.

Setting the **Subscribed** property changes the subscription status of the current mailbox. Setting the property to True adds the mailbox to the user's list of subscribed mailboxes, while setting it to False removes the mailbox from the subscription list.

Data Type

Boolean

See Also

MailboxName Property, SubscribeMailbox Method, UnsubscribeMailbox Method

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
ImapClient1.ThrowError = False
nError = ImapClient1.Connect(strHostName)

If nError > 0 Then
    MsgBox ImapClient1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

ImapClient1.ThrowError = True
ImapClient1.Connect strHostName

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error.

Note that the **Timeout** property also determines the amount of time the control will spend attempting to connect to a server. If a connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing 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
imapTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
imapTraceError	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.
imapTraceWarning	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.
imapTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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

UnreadMessages Property

Returns the number of unread messages in the current mailbox.

Syntax

object.**UnreadMessages**

Remarks

The **UnreadMessages** property returns the number of messages which do not have the SEEN flag in the current mailbox. This value is not the same as the number of recent messages in a mailbox, which is based on when the message was stored in the mailbox. To obtain a list of messages that have not been read, use the **SearchMailbox** method with UNSEEN as the search criteria.

It is possible that a message may be flagged as seen if it has been previously accessed by a different mail client. For example, a client may retrieve a message from an INBOX mailbox using the POP3 protocol, which would cause that message to be flagged as seen. This behavior is server dependent, and is most commonly found where the mail server supports both the POP3 and IMAP4 protocols.

Data Type

Integer (Int32)

See Also

MessageCount Property, NewMessages Property, RecentMessages Property, SearchMailbox Method

UserName Property

Gets and sets the current user name.

Syntax

object.**UserName** [= *username*]

Remarks

The **UserName** property specifies the user that is logging in to the server, and is required for authentication purposes. This property is used as the default value for the **Connect** method if no password is specified as an argument.

Data Type

String

See Also

AuthType Property, BearerToken Property, Password Property, Connect Method

Version Property

Return the current version of the object.

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

Internet Message Access Protocol Control Methods

Method	Description
Cancel	Cancels the current blocking network operation
CheckMailbox	Create a checkpoint of the currently selected mailbox
CloseMessage	Closes the current message
Command	Send a custom command to the server
Connect	Establish a connection with a server
CopyMessage	Copy a message from the current mailbox to another mailbox
CreateMailbox	Creates a new mailbox on the server
CreateMessage	Create a new message
DeleteMailbox	Deletes a mailbox from the server
DeleteMessage	Marks a message for deletion from the current mailbox
Disconnect	Terminate the connection with a server
ExamineMailbox	Selects the specified mailbox for read-only access
GetHeader	Returns the value of a header field from the specified message part
GetHeaders	Retrieves the headers for the specified message from the server
GetMessage	Retrieve a message from the server
Idle	Enables mailbox status monitoring for the client session
Initialize	Initialize the control and validate the runtime license key
OpenMessage	Open a message on the server
Read	Return data read from the server
Refresh	Updates the list of available mailboxes
RenameMailbox	Change the name of a mailbox
ReselectMailbox	Reselects the current mailbox
Reset	Reset the internal state of the control
SearchMailbox	Search the current mailbox for messages that match the specified criteria
SelectMailbox	Selects the specified mailbox for read-write access
StoreMessage	Retrieve a message from the current mailbox and store it in a local file
SubscribeMailbox	Subscribes the user to the specified mailbox
UndeleteMessage	Removes the deletion flag for the specified message
Uninitialize	Uninitialize the control and release any system resources that were allocated
UnselectMailbox	Unselects the current mailbox
UnsubscribeMailbox	Unsubscribes the user from the specified mailbox
Write	Write data to the server

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

Disconnect Method, Reset Method, OnCancel Event

CheckMailbox Method

Create a checkpoint of the currently selected mailbox.

Syntax

object.**CheckMailbox**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CheckMailbox** method requests that the server create a checkpoint of the currently selected mailbox, and updates the current number of new, unread messages available to the client.

When the client requests a checkpoint, the server may perform implementation-dependent housekeeping for that mailbox, such updating the mailbox on disk with the current state of the mailbox in memory. On some systems this command has no effect other than to update the client with the current number of messages in the mailbox.

This function actually sends two IMAP commands. The first is the CHECK command, followed by the NOOP command to poll for any new messages that have arrived. In addition to polling the server for new messages, this command can also be used to ensure the idle timer on the server does not expire and force a disconnect from the client.

See Also

MessageCount Property, NewMessages Property, RecentMessages Property, Command Method

CloseMessage Method

Closes the current message.

Syntax

object.**CloseMessage**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CloseMessage** method closes the current message. If there is any remaining data left to be read from the message, it will be read and discarded.

See Also

OpenMessage Method, Read Method

Command Method

Send a custom command to the server.

Syntax

object.**Command**(*Command*, [*Parameters*], [*Options*])

Parameters

Command

A string which specifies the command to send. Valid commands vary based on the Internet protocol and the type of server that the client is connected to. Consult the protocol standard and/or the technical reference documentation for the server to determine what commands may be issued by a client application.

Parameters

An optional string which specifies one or more parameters to be sent along with the command. If more than one parameter is required, most Internet protocols require that they be separated by a single space character. Consult the protocol standard and/or technical reference documentation for the server to determine what parameters should be provided when issuing a specific command. If no parameters are required for the command, this argument may be omitted.

Options

A numeric value which specifies one or more options. Currently this argument is reserved and should either be omitted, or a value of zero should always be used.

Return Value

A value of zero is returned if the command was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure. To determine the result code returned by the server in response to the command, read the value of the **ResultCode** property.

Remarks

The **Command** method sends a command to the server and processes the result code sent back in response to that command. This method can be used to send custom commands to a server to take advantage of features or capabilities that may not be supported internally by the control.

See Also

ResultCode Property, ResultString Property, OnCommand Event

Connect Method

Establish a connection with a server.

Syntax

object.**Connect**([*RemoteHost*], [*RemotePort*], [*UserName*], [*Password*], [*Timeout*], [*Options*])

Parameters

RemoteHost

A string which specifies the host name or IP address of the server. If this argument is not specified, it defaults to the value of the **HostAddress** property if it is defined. Otherwise, it defaults to the value of the **HostName** property.

RemotePort

A number which specifies the port to connect to on the server. If this argument is not specified, it defaults to the value of the **RemotePort** property. A value of zero indicates that the default port number for this service should be used to establish the connection. If the secure port number is specified, an implicit TLS connection will be established by default.

UserName

A string which specifies the name of the user used to authenticate access to the server. If this argument is not specified, it defaults to the value of the **UserName** property.

Password

A string which specifies the password used to authenticate the user. If you are using an OAuth 2.0 authentication method, this property should specify the access token provided by the mail service and not the user password. Refer to the **AuthType** property for more information about the supported authentication methods. If this argument is not specified, it defaults to the value of the **BearerToken** or **Password** property, depending on the authentication method specified.

Timeout

The number of seconds that the client will wait for a response before failing the operation. If this argument is not specified, the value of the **Timeout** property will be used as the default.

Options

A numeric value which specifies one or more options. If this argument is omitted or a value of zero is specified, a default connection will be established. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description
imapOptionNone	No connection options specified. A standard connection to the server will be established using the specified host name and port number.
imapOptionIdentify	This option specifies the client should identify itself to the server. If enabled, the client will send the ID command to the server as

	defined in RFC 2971. This option has no effect if the server does not support the ID command.	
&H400	imapOptionTunnel	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
&H800	imapOptionTrustedSite	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 the TLS protocol.
&H1000	imapOptionSecureExplicit	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. This option initiates the secure session using the STARTTLS command.
&H2000	imapOptionSecureImplicit	This option specifies the client should attempt to establish a secure connection with the server. The server must support secure connections using the TLS protocol, and the secure session must be negotiated immediately after the connection has been established.
&H8000	imapOptionSecureFallback	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.
&H40000	imapOptionPreferIPv6	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.

See Also

AuthType Property, BearerToken Property, HostAddress Property, HostName Property, Options Property, RemotePort Property, Disconnect Method, OnConnect Event

CopyMessage Method

Copy a message from the current mailbox to another mailbox.

Syntax

object.**CopyMessage**(*MessageNumber*, *MailboxName*, [*Options*])

Parameters

MessageNumber

The message identifier which specifies which message is to be copied to the mailbox. This value must be greater than zero and specify a valid message number.

MailboxName

A string which specifies the name of the mailbox that the message will be copied to. The mailbox must already exist, and the client must have the appropriate access rights to modify the mailbox.

Options

An optional parameter reserved for future use. This argument should either be omitted, or always be zero.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CopyMessage** method copies a message from the current mailbox to the specified mailbox. The message is appended to the mailbox, and the message flags and internal date are preserved. If the mailbox does not exist, the function will fail. To create a new mailbox, use the **CreateMailbox** method. A message can be copied within the same mailbox, in which case the server may flag it as a new message.

See Also

CreateMailbox Method, CreateMessage Method, GetMessage Method

CreateMailbox Method

Creates a new mailbox on the server.

Syntax

object.**CreateMailbox**(*MailboxName*)

Parameters

MailboxName

A string which specifies the name of the new mailbox to be created.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

If the mailbox name is suffixed with the server's hierarchy delimiter, this indicates to the server that the client intends to create mailbox names under the specified name in the hierarchy. If superior hierarchical names are specified in the mailbox name, then the server may automatically create them as needed. For example, if the mailbox name "Mail/Office/Projects" is specified and "Mail/Office" does not exist, it may be automatically created by the server.

The special mailbox name INBOX is reserved, and cannot be created. It is recommended that mailbox names only consist of printable ASCII characters, and the special characters "*" and "%" should be avoided.

See Also

CheckMailbox Method, DeleteMailbox Method, ExamineMailbox Method, SelectMailbox Method

CreateMessage Method

Create a new message.

Syntax

object.**CreateMessage**(*MessageData*, [*MessageFlags*], [*MailboxName*])

Parameters

MessageData

The contents of the message to be created. This may either be specified as a string or as an array of bytes.

MessageFlags

An optional integer value which specifies one or more message flags. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
imapFlagNone	No value.
imapFlagAnswered	The message has been answered.
imapFlagDraft	The message is not completed and is marked as a draft copy.
imapFlagUrgent	The message is flagged for urgent or special attention.
imapFlagSeen	The message has been read.

MailboxName

An optional string argument which specifies the name of the mailbox that the message will be created in. If this argument is omitted, the message will be created in the currently selected mailbox.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CreateMessage** method creates a new message, appending it to the contents of the specified mailbox. This method will cause the current thread to block until the message transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

GetMessage Method, OnProgress Event

DeleteMailbox Method

Deletes a mailbox from the server.

Syntax

object.**DeleteMailbox**(*MailboxName*)

Parameters

MailboxName

A string which specifies the name of the mailbox to be deleted.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

A mailbox cannot be deleted if it contains inferior hierarchical names and has the **imapFlagNoSelect** attribute. On most systems this is the case when the mailbox name references a directory on the server, and that directory contains other subdirectories or mailboxes. To remove the mailbox, you must first delete any child mailboxes that exist.

If the mailbox that is deleted is the currently selected mailbox, it will be automatically unselected and any messages marked for deletion will be expunged before the mailbox is removed. If the delete operation fails, the client will remain in an unselected state until either the **ExamineMailbox** or **SelectMailbox** method is called.

The special mailbox name INBOX is reserved, and cannot be deleted.

See Also

CheckMailbox Method, CreateMailbox Method, ExamineMailbox Method, SelectMailbox Method

DeleteMessage Method

Marks a message for deletion from the current mailbox.

Syntax

object.**DeleteMessage**(*MessageNumber*)

Parameters

MessageNumber

Number of message to delete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **DeleteMessage** method only flags the message for deletion. The message is not actually deleted until the mailbox is expunged or another mailbox is selected. This function will return an error if the current mailbox is in read-only mode, such as if it was selected using the **ExamineMailbox** method.

It is important to note that unlike the POP3 protocol, a message that is marked for deletion is still accessible on the IMAP server until the mailbox is expunged. This means, for example, that a deleted message can still be retrieved using the **GetMessage** method.

To determine if a message has been marked for deletion, set the **Message** property to the message number and then check the value of the **MessageFlags** property to determine if the **imapFlagDeleted** bit flag has been set.

To remove the deletion flag from the message, use the **UndeleteMessage** method. To prevent all messages in the current mailbox from being expunged, use the **ReselectMailbox** function to reset the current mailbox state. Calling the **Reset** method will also unselect the current mailbox without expunging deleted messages.

See Also

MessageFlags Property, CopyMessage Method, CreateMessage Method, ExamineMailbox Method, ReselectMailbox Method, Reset Method, UndeleteMessage Method

Disconnect Method

Terminate the connection with a server.

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

IsConnected Property, Connect Method, OnDisconnect Event

ExamineMailbox Method

Selects the specified mailbox for read-only access.

Syntax

object.**ExamineMailbox**(*MailboxName*)

Parameters

MailboxName

A string argument which specifies the name of the mailbox to be examined.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ExamineMailbox** method is used to select a mailbox in read-only mode. Messages can be read, but they cannot be modified or deleted from the mailbox and new messages will not lose their status as new messages if they are accessed.

If the client has a different mailbox currently selected, that mailbox will be closed and any messages marked for deletion will be expunged. To prevent deleted messages from being removed from the previous mailbox, use the **UnselectMailbox** method prior to examining the new mailbox.

The special case-insensitive mailbox name INBOX is used for new messages. Other mailbox names may or may not be case-sensitive depending on the IMAP server's operating system and implementation.

To access a mailbox in read-write mode, use the **SelectMailbox** method.

See Also

MailboxName Property, CheckMailbox Method, ReselectMailbox Method, SelectMailbox Method

GetHeader Method

Returns the value of a header field from the specified message part.

Syntax

object.**GetHeader**(*MessageNumber*, *MessagePart*, *HeaderField*, *HeaderValue*)

Parameters

MessageNumber

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

MessagePart

The message part that the header value will be be retrieved from. Message parts start with a value of one. A value of zero specifies that the RFC822 header field for the message will be used.

HeaderField

A string which specifies the message header to retrieve. The colon should not be included in this string.

HeaderValue

A string variable which will contain the value of the specified message header if the method is successful.

Return Value

A value of true is returned if the header was present and could be retrieved, otherwise a value of false is returned.

Remarks

The **GetHeader** method returns the value of a header field from the specified message part. This allows an application to be able to easily determine the value of a header such as the sender, or the subject of the message. Any header field, including non-standard extensions, may be returned by this method.

See Also

HeaderField Property, HeaderValue Property, GetMessage Method

GetHeaders Method

Retrieves the headers for the specified message from the server.

Syntax

object.**GetHeaders**(*MessageNumber*, *Headers*)

Parameters

MessageNumber

Number of article to retrieve from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Headers

A string or byte array which will contain the data transferred from the server when the method returns.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **GetHeaders** method is used to retrieve the message headers from the server and copy it into a local buffer. This method will cause the current thread to block until the article transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

Note that the header data will be from the first part of the message, not from any additional sections of a multipart message. In other words, the headers such as From, To, Subject and Date will be returned in the buffer. To retrieve the headers from a specific section of a multipart message, you can use the **GetMessage** method and specify the **imapSectionHeader** option.

See Also

MessageCount Property, CreateMessage Method, GetMessage Method, OpenMessage Method, OnProgress Event

GetMessage Method

Retrieve a message from the server.

Syntax

object.**GetMessage**(*MessageNumber*, *MessagePart*, *MessageData*, [*Options*])

Parameters

MessageNumber

Number of message to retrieve from the server. This value must be greater than zero. The first message in the mailbox is message number one.

MessagePart

The message part that will be retrieved. A value of zero specifies that the complete message should be returned. If the message is a multipart MIME message, message parts start with a value of one.

MessageData

A string or byte array which will contain the data transferred from the server when the method returns.

Options

An optional integer value which specifies one or more options. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description
imapSectionDefault	All headers and the complete body of the specified message or message part are to be retrieved. The client application is responsible for parsing the header block. If the message is a MIME multipart message and the complete message is returned, the application is responsible for parsing the individual message parts if necessary.
imapSectionHeader	All headers for the specified message or message part are to be retrieved. The client application is responsible for parsing the header block.
imapSectionMimeHeader	The MIME headers for the specified message or message are to be retrieved. Only those header fields which are used in MIME messages, such as Content-Type will be returned to the client. This is typically useful when processing the header for a multipart message which contains file attachments. The client application is responsible for parsing the header block.
imapSectionBody	The body of the specified message or message part is to be retrieved. For a MIME formatted message, this may include data that is uuencoded or base64 encoded. The application is responsible for decoding this data.
imapSectionPreview	The message header or body is being previewed and should not be marked as read by the server. This prevents the message from having the **imapFlagSeen** flag from being automatically set when the message data is retrieved.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **GetMessage** method is used to retrieve a message from the server and copy it into a local buffer. This method will cause the current thread to block until the message transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

CopyMessage Method, CreateMessage Method, DeleteMessage Method, OpenMessage Method, OnProgress Event

Idle Method

Enables mailbox status monitoring for the client session.

Syntax

object.**Idle**([*Options*], [*Timeout*])

Parameters

Options

An optional integer value which specifies how the **Idle** method will function. If this argument is omitted, the method will return immediately to the caller without causing the current thread to block.

Value	Description
imapIdleNoWait	The method should return immediately after idle processing has been enabled. When this option is used, the application may continue to perform other functions while the client session is monitored for status updates sent by the server. The client will continue to monitor status changes until an IMAP command issued or the client disconnects from the server. This is the default option.
imapIdleWait	The method should wait until the server sends a status update, or until the timeout period is reached. The client will stop monitoring status changes when the function returns. If this option is used in a single-threaded application, normal message processing can be impeded, causing the application to appear non-responsive until the timeout period is reached. It is strongly recommended that single-threaded applications with a user interface specify the **imapIdleNoWait** option instead.

Timeout

Specifies the timeout period in seconds to wait for a notification from the server. This parameter is only used when the **imapIdleWait** option has been specified.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Idle** method enables mailbox status monitoring for the client session, allowing the client to receive notifications from the server whenever a new message arrives or a message is expunged from the currently selected mailbox. This is typically used as an alternative to the client periodically polling the server for status information.

Many IMAP servers support the ability to asynchronously send status updates to the client, rather than have the client periodically poll the server. The client enables this feature by calling the **Idle** method and implementing an event handler for the **OnUpdate** event. Typically these events inform the client that a new message has arrived or that a message has been expunged from the mailbox.

The **Idle** method can operate in two modes, based on the options specified by the caller. If the option **imapIdleNoWait** is specified, the method begins monitoring the client session asynchronously and returns control immediately to the caller. If the server sends a update notification to the client, the

OnUpdate event will fire with information about the status change. If the option **imapIdleWait** is specified, the method will block waiting for the server to send a notification message to the client. The method will return when either a message is received or the timeout period is exceeded.

Sending an IMAP command to the server will cause the client to stop monitoring the session for status changes. To explicitly stop monitoring the session, use the **Cancel** method.

This method works by sending the IDLE command to the server and starting a worker thread which monitors the connection and looks for untagged responses issued by the server. Events will be generated for EXISTS, EXPUNGE and RECENT messages. Note that some servers may periodically send untagged OK messages to the client, indicating that the connection is still active; these messages are explicitly ignored.

An application should never make an assumption about how a particular server may send update notifications to the client. Servers can be configured to use different intervals at which notifications are sent. For example, a server may send new message notifications immediately, but may periodically notify the client when a message has been expunged. Alternatively, a server may only send notifications at fixed intervals, in which case the client would not be notified of any new messages until the interval period is reached. It is not possible for a client to know what a particular server's update interval is. Applications that require that degree of control should not use the **Idle** method and should poll the server instead.

See Also

OnUpdate Event

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 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 imapClient = CreateObject("SocketTools.ImapClient.11")

nError = imapClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize Method

OpenMessage Method

Open a message on the server.

Syntax

object.**OpenMessage**([*MessageNumber*], [*MessagePart*], [*Offset*], [*Length*], [*Options*])

Parameters

MessageNumber

Number of message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one. If this argument is omitted, the current message selected by the **Message** property will be opened.

MessagePart

The message part that will be retrieved. A value of zero specifies that the complete message should be returned. If the message is a multipart MIME message, message parts start with a value of one. If this argument is omitted, the complete contents of the message will be accessible.

Offset

The byte offset into the message. This parameter can be used in conjunction with the ***Length*** argument to return a specific part of a message. If this argument is omitted or a value of zero is specified, the server will return data from the beginning of the message.

Length

An optional integer value which specifies the maximum number of bytes the client wishes to read. To specify the entire message, from the offset specified by the ***Offset*** argument to the end of the message, use a value of -1. If this argument is omitted, the entire contents of the message will be returned, starting at the byte offset specified by the ***Offset*** argument. If both the ***Offset*** and ***Length*** arguments are omitted, the entire contents of the message will be returned.

Options

An optional integer value which specifies one or more options. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description
imapSectionDefault	All headers and the complete body of the specified message or message part are to be retrieved. The client application is responsible for parsing the header block. If the message is a MIME multipart message and the complete message is returned, the application is responsible for parsing the individual message parts if necessary.
imapSectionHeader	All headers for the specified message or message part are to be retrieved. The client application is responsible for parsing the header block.
imapSectionMimeHeader	The MIME headers for the specified message or message are to be retrieved. Only those header fields which are used in MIME messages, such as Content-Type will be returned to the client. This is typically useful when processing the header for a multipart message which contains file attachments. The client application is responsible for parsing the header block.

imapSectionBody	The body of the specified message or message part is to be retrieved. For a MIME formatted message, this may include data that is uuencoded or base64 encoded. The application is responsible for decoding this data.
imapSectionPreview	The message header or body is being previewed and should not be marked as read by the server. This prevents the message from having the **imapFlagSeen** flag from being automatically set when the message data is retrieved.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **OpenMessage** method opens a message or a specific part of a multipart message in the current mailbox. The message data may also be limited a specific byte offset and length, which can be useful for previewing the contents. The client can then read the contents of the message using the **Read** method, and once all of the data has been read, the message should be closed by calling the **CloseMessage** method.

See Also

CloseMessage Method, GetMessage Method, Read Method

Read Method

Return data read from the server.

Syntax

object.**Read**(*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.

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 server 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 server, 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.

See Also

IsConnected Property, IsReadable Property, Write Method, OnRead Event, OnWrite Event

Refresh Method

Updates the list of available mailboxes.

Syntax

object.**Refresh**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Refresh** method updates the list of mailboxes that may be selected by the client. The available mailboxes may be enumerated using the **Mailbox** property array, with the **Mailboxes** property returning the total number of mailboxes.

See Also

Mailbox Property, Mailboxes Property, MailboxName Property, ExamineMailbox Method, RenameMailbox Method, ReselectMailbox Method, SelectMailbox Method

RenameMailbox Method

Change the name of a mailbox.

Syntax

object.**RenameMailbox**(*OldName*, *NewName*)

Parameters

OldName

A string that specifies the name of the mailbox to be renamed on the server. The mailbox must exist on the server, otherwise an error will be returned.

NewName

A string that specifies the new name for the mailbox. An error will be returned if a mailbox with that name already exists.

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

If the existing mailbox name contains inferior hierarchical names (mailboxes under the specified mailbox) then those mailboxes will also be renamed. For example, if the mailbox "Mail/Pictures" contains two mailboxes, "Personal" and "Work" and it is renamed to "Mail/Images" then the two mailboxes under it would be automatically renamed to "Mail/Images/Personal" and "Mail/Images/Work".

If the mailbox being renamed is the currently selected mailbox, the current mailbox will be unselected and any messages marked for deletion will be expunged. The new mailbox name will then automatically be re-selected. To prevent deleted messages from being removed from the mailbox prior to being renamed, use the **UnselectMailbox** method to unselect the current mailbox before calling **RenameMailbox**. Note that if the rename operation fails, the client may be left in an unselected state.

It is permitted to rename the special mailbox INBOX. In this case, the messages will be moved from the INBOX mailbox to the new mailbox. If the INBOX mailbox is currently selected, the new mailbox will not automatically be selected. INBOX will remain the selected mailbox.

See Also

Mailbox Property, Mailboxes Property, MailboxName Property, ExamineMailbox Method, ReselectMailbox Method, SelectMailbox Method

ReselectMailbox Method

Reselects the current mailbox.

Syntax

object.**ReselectMailbox**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ReselectMailbox** method forces the current mailbox to be reselected and updates those properties which return information about the mailbox, such as the **MailboxFlags** property. Deleted messages are not expunged from the mailbox and remain marked for deletion.

See Also

MailboxName Property, MailboxFlags Property, MailboxUID Property, ExamineMailbox Method, SelectMailbox 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, open network connections will be closed and any handles allocated by the control will be released.

See Also

Cancel Method, Initialize Method, Uninitialize Method

SearchMailbox Method

Search the current mailbox for messages that match the specified criteria.

Syntax

object.**SearchMailbox**(*Criteria*, *Messages*, [*CharacterSet*])

Parameters

Criteria

A string which consists of one or more keywords which are used to define the search criteria. The following keywords are recognized:

Keyword	Description
ANSWERED	Match those messages which have the **imapFlagAnswered** flag set.
BCC *address*	Match those messages which contain the specified address in the BCC header field.
BEFORE *date*	Match those messages which were added to the mailbox prior to the specified date.
BODY *string*	Match those messages where the body contains the specified string.
CC *address*	Match those messages which contain the specified address in the CC header field.
DELETED	Match those messages which have the **imapFlagDeleted** flag set.
DRAFT	Match those messages which have the **imapFlagDraft** flag set.
FLAGGED	Match those messages which have the **imapFlagUrgent** flag set.
FROM *address*	Match those messages which contain the specified address in the FROM header field.
HEADER *field string*	Match those messages which contain the string in the specified header field. If no string is specified, then all messages which contain the header will be matched.
LARGER *size*	Match those messages which are larger than the specified size in bytes.
NEW	Match those messages which have the **imapFlagRecent** flag set, but not the **imapFlagSeen** flag.
OLD	Match those messages which do not have the **imapFlagRecent** flag set.
ON *date*	Match those messages which were added on the specified date.
RECENT	Match those messages which have the **imapFlagRecent** flag set.
SEEN	Match those messages which have the **imapFlagSeen** flag set.
SENTBEFORE *date*	Match those messages whose Date header value is earlier than the specified date.
SENTON *date*	Match those messages whose Date header value is the same as the specified date.
SENTSINCE	Match those messages whose Date header value is later than the

date	specified date.
SINCE *date*	Match those messages added to the mailbox after the specified date.
SMALLER *size*	Match those messages which are smaller than the specified size in bytes.
SUBJECT *string*	Match those messages whose Subject header contains the specified string.
TEXT *string*	Match those messages whose headers or body contains the specified string.
TO *address*	Match those messages which contain the specified address in the TO header field.
UID *sequence*	Match those messages with unique identifiers in the sequence set.
UNANSWERED	Match those messages which do not have the **imapFlagAnswered** flag set.
UNDELETED	Match those messages which do not have the **imapFlagDeleted** flag set.
UNDRAFT	Match those messages which do not have the **imapFlagDraft** flag set.
UNFLAGGED	Match those messages which do not have the **imapFlagUrgent** flag set.
UNSEEN	Match those messages which do not have the **imapFlagSeen** flag set.

Messages

This argument must be passed as an array of integers which will contain the message numbers of those messages which match the search criteria. The size of the array determines the maximum number of matches that will be returned by the method. Note that the array must specify 32-bit integers. In Visual Basic, this means that the array would be typed as Long. In Visual Basic.NET, the array would be typed as Integer.

CharacterSet

An optional string which specifies the character set to use when searching the mailbox. If this argument is omitted, the default UTF-8 character set will be used. Note that not all servers support searches using anything but the default character set.

Return Value

This method will return the number of messages which were found to match the search criteria. If no messages match the criteria, then the return value will be zero. A return value of -1 indicates an error, and the specific error code can be determined by checking the **LastError** property.

Remarks

The **SearchMailbox** method is used to search a mailbox for messages which match a given criteria and return a list of the matching message numbers. The search criteria is composed of one or more search keywords and and optional value to match against. String searches are not case sensitive and partial matches in the message are returned. The message numbers returned by this method are only valid until the mailbox is expunged or another mailbox is selected.

In addition to the listed keywords, the keyword NOT may prefix a keyword to return those messages which do not match the search criteria. For example, "NOT TO user@domain.com" would return those messages which were not addressed to user@domain.com.

If multiple search keywords are specified, the result is the intersection of all those messages which meet the search criteria. For example, a search criteria of "DELETED SINCE 1-Jan-2010" would return

all those messages which are marked for deletion and were added to the mailbox after 1 January 2010.

Those search keywords which expect dates must be specified in format ***dd-mmm-yyyy*** where the month is the three letter abbreviation for the month name. Note that the internal date the message was added to the mailbox is not the same as the value of the Date header field in the message.

If the search keyword expects a string value and the string contains one or more spaces, you need to enclose the search string in quotes as part of the criteria string. For example, in Visual Basic you could use code like this:

```
strCriteria = "SUBJECT " + Chr(34) + "search string" + Chr(34)
```

The quotes around the search string prevents the server from interpreting it as a multiple search criteria to be evaluated. If you are using a search string provided by a user, it is recommended that you always enclose it in quotes to prevent any potential ambiguity in the search. Even if the search string does not contain any spaces, it is always safe to enclose it in quotes.

The UID keyword expects a one or more unique message identifiers. These values may provided as comma separated list, or a range delimited by a colon. For example, "UID 23000:24000" would return all those messages who have UIDs ranging from 23000 through to 24000.

See Also

MailboxName Property, ExamineMailbox Method, SelectMailbox Method

SelectMailbox Method

Selects the specified mailbox for read-write access.

Syntax

object.**SelectMailbox**(*MailboxName*)

Parameters

MailboxName

A string argument which specifies the name of the mailbox to be selected.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **SelectMailbox** method is used to select a mailbox in read-write mode. If the client has a different mailbox currently selected, that mailbox will be closed and any messages marked for deletion will be expunged. To prevent deleted messages from being removed from the previous mailbox, use the **UnselectMailbox** method prior to selecting the new mailbox.

The special case-insensitive mailbox name INBOX is used for new messages. Other mailbox names may or may not be case-sensitive depending on the IMAP server's operating system and implementation.

To access a mailbox in read-only mode, use the **ExamineMailbox** method.

See Also

MailboxName Property, CheckMailbox Method, ExamineMailbox Method, ReselectMailbox Method

StoreMessage Method

Retrieve a message from the current mailbox and store it in a local file.

Syntax

object.**StoreMessage**(*MessageNumber*, *FileName*)

Parameters

MessageNumber

Number of message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

FileName

A string which specifies the file that the message will be stored in. If the file does not exist, it will be created. If the file does exist, it will be overwritten with the contents of the message.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **StoreMessage** method retrieves a message from the server and stores it in a file on the local system. The contents of the message is stored as a text file, using the specified file name. This method will cause the current thread to block until the message transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

GetMessage Method, OpenMessage Method

SubscribeMailbox Method

Subscribes the user to the specified mailbox.

Syntax

object.**SubscribeMailbox**(*MailboxName*)

Parameters

MailboxName

A string which specifies the name of the mailbox to subscribe to.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **SubscribeMailbox** method adds the specified mailbox to the current user's list of active or subscribed mailboxes. The user will remain subscribed to the mailbox across multiple sessions, until the **UnsubscribeMailbox** method is called to remove the mailbox from the subscription list.

Note that if a user subscribes to a mailbox and that mailbox is later renamed or deleted, the mailbox will not be automatically removed from the user's subscription list. To determine if the current mailbox is in the user's subscription list, check the **Subscribed** property.

See Also

MailboxName Property, Subscribed Property, UnsubscribeMailbox Method

UndeleteMessage Method

Removes the deletion flag for the specified message.

Syntax

object.**UndeleteMessage**(*MessageNumber*)

Parameters

MessageNumber

Number of message to undelete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **UndeleteMessage** method removes the deletion flag for the specified message in the current mailbox. To determine if a message has been marked for deletion, set the **Message** property to the message number and then check the value of the **MessageFlags** property to determine if the **imapFlagDeleted** bit flag has been set.

See Also

MessageFlags Property, DeleteMessage Method, ReselectMailbox 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. 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

UnselectMailbox Method

Unselects the current mailbox.

Syntax

object.**UnselectMailbox**([*Expunge*])

Parameters

Expunge

An optional boolean argument which determines if deleted messages will be expunged from the mailbox. A value of true specifies that messages that have been marked for deletion will be removed from the mailbox. A value of False specifies that no messages will be removed from the mailbox. If this argument is omitted, the default action is to expunge deleted messages from the mailbox.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **UnselectMailbox** method unselects the current mailbox. Once the mailbox has been unselected, no messages in that mailbox can be accessed, and by default any messages which have been marked for deletion are removed.

See Also

MailboxName Property, ExamineMailbox Method, SelectMailbox Method

UnsubscribeMailbox Method

Unsubscribes the user from the specified mailbox.

Syntax

object.**UnsubscribeMailbox**(*MailboxName*)

Parameters

MailboxName

A string which specifies the name of the mailbox to unsubscribe from.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **UnsubscribeMailbox** method removes the specified mailbox from the current user's list of active or subscribed mailboxes. To determine if the current mailbox is in the user's subscription list, check the **Subscribed** property.

See Also

MailboxName Property, Subscribed Property, SubscribeMailbox Method

Write Method

Write data to the server.

Syntax

object.**Write**(*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 because 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 you wish to send the data without conversion, use a **Byte** array as the buffer instead of a **String** variable.

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 server, or -1 if an error was encountered.

Remarks

The **Write** method sends the data in ***buffer*** to the server. 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 blocking, the application will wait until the data can be sent. If the control is non-blocking and the write fails because it could not send all of the data to the server, the **OnWrite** event will be fired when the server can accept data again.

See Also

IsConnected Property, IsWritable Property, Timeout Property, Read Method, OnWrite Event

Internet Message Access Protocol Control Events

Event	Description
OnCancel	This event is generated when a blocking operation is canceled
OnConnect	This event is generated when a connection is established
OnDisconnect	This event is generated when a connection is terminated
OnError	This event is generated when a control error occurs
OnProgress	This event is generated during data transfer
OnRead	This event is generated when data is available to be read
OnTimeout	This event is generated when a blocking operation times out
OnUpdate	This event is generated when the server sends a mailbox update notification to the client
OnWrite	This event is generated when data can be written to the server

OnCancel Event

The **OnCancel** event is generated when a blocking operation is canceled.

Syntax

Sub *object*_**OnCancel** ([*Index* **As Integer**])

Remarks

This event is generated 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, OnTimeout Event

OnCommand Event

The **OnCommand** event is generated when the client sends a command to the server and receives a reply indicating the results of that command.

Syntax

Sub *object*_**OnCommand**([*Index* **As Integer**], **ByVal** *ResultCode* **As Variant**, **ByVal** *ResultString* **As Variant**)

Remarks

The **OnCommand** event is generated when the client receives a reply from the server after some action has been taken. The ***ResultCode*** argument contains the numeric result code returned by the server. The result codes returned from the server fall into one of the following categories:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

The ***ResultString*** argument contains the descriptive string returned by the server which describes the result. The string contents may vary depending on the type of server.

See Also

ResultCode Property, ResultString Property, Command Method

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 server as a result of a **Connect** method call. This event is only triggered when the **Blocking** property is set to False.

See Also

Blocking Property, Connect Method, OnDisconnect Event, OnWrite 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. This event is only triggered when the **Blocking** property is set to False.

When the **OnDisconnect** event fires, it is possible that there may still be buffered data available to read from the server. Before disconnecting from the server, the application should attempt to read any remaining data until the **Read** method returns a value of zero, or returns an error indicating that the operation would block.

See Also

Blocking Property, IsConnected Property, IsReadable Property, Connect Method, Disconnect Method, Read Method, 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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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 during data transfer.

Syntax

Sub *object*_**OnProgress** ([*Index* **As Integer**], **ByVal** *MessageNumber* **As Variant**, **ByVal** *MessageSize* **As Variant**, **ByVal** *MessageCopied* **As Variant**, **ByVal** *Percent* **As Variant**)

Remarks

The **OnProgress** event is generated during the transfer of data between the client and server, indicating the amount of data exchanged. For transfers of 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:

MessageNumber

The number of the message that is being retrieved. If a message is being created, this argument will have a value of zero.

MessageSize

The size of the message being transferred in bytes. This value may be zero if the control cannot obtain the size of the message from the server.

MessageCopied

The number of bytes that have been transferred between the client and server.

Percent

The percentage of data that's been transferred, expressed as an integer value between 0 and 100, inclusive. If the size of the message on the server cannot be determined, this value will always be 100.

Note that this event is only generated when message data is transferred using the **CreateMessage**, **GetHeaders** or **GetMessage** methods. If the client is reading or writing the file data directly to the server using the **Read** or **Write** methods then the application is responsible for calculating the completion percentage and updating any user interface controls.

See Also

Message Property, MessagePart Property, CreateMessage Method, GetHeaders Method, GetMessage 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 triggered when the **Blocking** property is set to False.

See Also

IsReadable Property, 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. This event is only triggered when the **Blocking** property is set to True.

See Also

Timeout Property, OnCancel Event

OnUpdate Event

The **OnUpdate** event is generated when the server sends a mailbox update notification to the client.

Syntax

Sub *object*_**OnUpdate** ([*Index* **As Integer**], **ByVal** *UpdateType* **As Variant**, **ByVal** *MessageNumber* **As Variant**)

Remarks

The **OnUpdate** event is generated when the server sends a notification to the client that a new message has been stored in the mailbox, or when a message has been expunged from the mailbox. The arguments to this event are:

UpdateType

An integer value which specifies the type of update notification that has been sent by the server. It may be one of the following values:

Value	Description
imapUpdateUnknown	The server has sent an unrecognized notification message. The value of the ***MessageNumber*** argument is undefined for this type of notification. This does not necessarily reflect an error condition, as some servers may send additional notification messages beyond the standard EXISTS, EXPUNGE and RECENT messages. Most applications should ignore this type of notification.
imapUpdateMessage	The server has sent notification message to the client indicating that a new message has arrived. The ***MessageNumber*** argument will contain the message number for the new message. Typically this update notification occurs shortly after the new message has been stored in the current mailbox.
imapUpdateExpunge	The server has sent a notification message to the client indicating that a message has been removed from the current mailbox. The ***MessageNumber*** argument will contain the message number for the message that has been removed. It is recommended that the application re-examine the mailbox when this notification is received. Typically this notification is only sent periodically by the server, and may not be sent immediately after a message has been expunged from the mailbox.
imapUpdateMailbox	The server has sent notification message to the client indicating that the state of the mailbox has changed. The ***MessageNumber*** argument is not used with this notification. This message is sent periodically by the server and may not be sent immediately after a new message arrives or a message is flagged as unread. It is recommended that the application re-examine the mailbox when this notification is received.

MessageNumber

An integer value which specifies the message number associated with the status change. Note that this argument is not used with the **imapUpdateMailbox** notification and will contain a value of zero.

This event is only generated when the **Idle** method has been used to enable mailbox status monitoring.

See Also

Idle Method

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 triggered when the **Blocking** property is set to False.

See Also

IsWritable Property, Read Method, Write Method, OnConnect Event, OnRead Event

Internet Mail Control

Compose, send and retrieve messages using standard Internet email protocols.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	InternetMailCtl.InternetMail
File Name	CSIMCX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.InternetMail.11
ClassID	CA8D0AAF-27EA-4367-A43F-CCD18BAE026B
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 821, RFC 822, RFC 1034, RFC 1425, RFC 1869, RFC 1939 RFC 2045, RFC 2046, RFC 2047, RFC 2048, RFC 2821, RFC 3501

Overview

The Internet Mail ActiveX control had a comprehensive interface that provides everything required to incorporate email functionality in an application. The control implements the Simple Mail Transfer Protocol (SMTP) for sending messages, the Post Office Protocol (POP3) and Internet Message Access Protocol (IMAP) for retrieving messages from a mail serve. The Multipurpose Internet Mail Extensions (MIME) standard is implemented for composing and processing messages.

Many of the control's properties control the contents of a message, such as the list of recipients, the subject of the message and the message body. Methods are used to compose new messages, retrieve messages from a mail server and deliver messages to one or more recipients. Messages can also be managed on the mail server, or downloaded to the local system and stored in a file or a database record. The application has complete access to all of the headers in the message, and can create custom application-specific header fields if needed. Event notifications enable the application to provide the user with feedback, such as the progress of sending or retrieving a message. Additional features such as delivery status notification, support for relay servers and secure encrypted connections are easily implemented by setting a property or specifying an option when calling a method.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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 Mail Control Properties

Property	Description
AllHeaders	Returns the complete RFC 822 header values for the current message
AllRecipients	Returns a comma-separated list of all message recipients
Attachment	Return the name of the attached file in the current message part
Bcc	Gets and sets the list of addresses that should receive a blind copy of the current message
BearerToken	Gets and sets the OAuth 2.0 bearer token used for authentication
Cc	Gets and sets the list of addresses that should receive a copy of the current message
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 client certificate
CertificatePassword	Gets and sets the password associated with the client certificate
CertificateStatus	Return the status of the server certificate
CertificateStore	Gets and sets the name of the client 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
ContentID	Gets and sets the content identifier for the selected message part
ContentLength	Returns the size of the data stored in the selected message part
ContentType	Gets and sets the content type of the selected message part
Date	Gets and sets the date for the current message
Domain	Gets and sets the local domain name
Encoding	Gets and sets the content encoding for the current message part
From	Gets and sets the address of the person who sent the message
HashStrength	Return the length of the message digest that was selected
IsBlocked	Determine if the control is blocked performing an operation
IsConnected	Determine if the control is connected to a server
IsInitialized	Determine if the control has been initialized
IsInitialized	Determine if the control has been initialized
LastError	Gets and sets the last error code
LastErrorString	Return a description of the last error that occurred
LastMessage	Return the number of the last message available on the server
Localize	Enable or disable message localization
Mailbox	Returns the name of the specified mailbox from a list of mailboxes on the server
Mailboxes	Returns the number of mailboxes available on the server
MailboxFlags	Returns one or more flags which identify characteristics of the current mailbox
MailboxName	Gets and sets the name of the current mailbox
MailboxSize	Return the size of the current mailbox in bytes
MailboxUID	Returns the unique identifier for the current mailbox
Mailer	Gets and sets the name of the mailer application
Message	Gets and sets the current message headers and text

MessageCount	Return the number of messages available on the server
MessageFlags	Returns one or more flags which identify characteristics of the current message
MessageID	Return a unique identifier for the current message
MessageIndex	Gets and sets the current message number on the server
MessagePart	Gets and sets the current part in a multipart message
MessageParts	Return the number of parts in the current message
MessageSize	Return the size of the current message in bytes
MessageText	Return or change the text in the current message part
MessageUID	Return the UID for the current message on the mail server
NameServer	Gets and sets the Internet address for a nameserver
NewMessages	Return the number of new messages available in the current mailbox
Organization	Return or change the name of the organization that created the message
Password	Gets and sets the password for the current user
Priority	Gets and sets the current message priority
RecentMessages	Returns the number of messages which have recently arrived in the mailbox
Recipient	Return the address of a message recipient
Recipients	Return the number of recipients for the current message
RelayServer	Gets and sets the host name or address of a relay server
RelayPort	Gets and sets the port number for the specified relay server
ReplyTo	Gets and sets the address of the person who should receive replies to this message
ReturnReceipt	Gets and sets the address of the person who should receive a message indicating that the message has been read
Secure	Specify if a connection to the server is secure
SecureCipher	Return the encryption algorithm used to establish the secure connection with the server
SecureHash	Return the message digest selected when establishing the secure connection with the server
SecureKeyExchange	Return the key exchange algorithm used to establish the secure connection with the server
SecureProtocol	Gets and sets the security protocol used to establish the secure connection with the server
ServerName	Gets and sets the host name of the current mail server
ServerPort	Gets and sets the port number for the current mail server
Subject	Gets and sets the subject of the current message
ThrowError	Enable or disable error handling by the container of the control
Timeout	Gets and sets the amount of time until a blocking network operation is aborted
TimeZone	Gets and sets the current timezone offset in seconds
To	Gets and sets the recipient of the current message
Trace	Enable or disable network function level tracing
TraceFile	Return or specify the network function trace output file
TraceFlags	Gets and sets the current network function tracing flags
UnreadMessages	Returns the number of unread messages in the current mailbox
UserName	Gets and sets the current user name
Version	Return the current version of the object

AllHeaders Property

Returns the complete RFC 822 header values for the current message.

Syntax

object.**AllHeaders**

Remarks

The **AllHeaders** property will return all of the RFC 822 header values in a string. This includes the message headers that are most commonly referred to, such as the To, From and Subject headers. Each header and its value are separated by a colon, and terminated with a carriage return and linefeed (CRLF) pair.

The headers and their values returned by this property will not be identical to the header block in the original message. If a header value is split across multiple lines, the text returned by this property will be folded, with the complete header value on a single line of text and removing any extraneous whitespace. If the header value has been encoded by the mail client, this property will return the decoded value, not the original encoded value.

Data Type

String

See Also

GetHeader, SetHeader

AllRecipients Property

Returns a comma-separated list of all message recipients.

Syntax

object.**AllRecipients**

Remarks

The **AllRecipients** property returns a string value that contains a comma-separated list of all message recipients. To individually enumerate through each of the recipient addresses, you can use the **Recipient** property array and **Recipients** property.

The string returned by this property will include those addresses specified by the **Bcc** property, even though they are not included in the message header.

Data Type

String

See Also

Bcc Property, Cc Property, Recipient Property, Recipients Property, To Property, ComposeMessage Method

Attachment Property

Return the name of the attached file in the current message part.

Syntax

object.**Attachment**

Remarks

The **Attachment** property returns the name of the file attachment in the current part of a multipart message. When a new part is selected that contains an attached file, the **Attachment** property is updated to reflect the attached file's name. If the current message part does not contain a file attachment, this property will return an empty string.

Data Type

String

See Also

AttachData Method, AttachFile Method, ExtractFile Method, FindAttachment Method

Bcc Property

Gets and sets the list of addresses that should receive a blind copy of the current message.

Syntax

object.**Bcc** [= *value*]

Remarks

The **Bcc** property is used to specify one or more addresses that a copy of the message will be delivered to. Note that these addresses are not included in the message header and cannot be viewed by the recipient.

Multiple addresses may be specified by separating them with a comma. Each address must conform to the standard Internet address format.

Data Type

String

See Also

Cc Property, From Property, ReplyTo Property, To Property

BearerToken Property

Gets and sets the OAuth 2.0 bearer token for the current user.

Syntax

object.**BearerToken** [= *token*]

Remarks

The **BearerToken** property specifies the OAuth 2.0 bearer token used to authenticate the user. Assigning a value to this property will change the current authentication method to use OAuth 2.0.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an bearer token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

If the current authentication method does not use OAuth 2.0, this property will return an empty string and you should check the value of the **Password** property to obtain the current user's password.

Data Type

String

See Also

Password Property, UserName Property, Connect Method

Cc Property

Gets and sets the list of addresses that should receive a copy of the current message.

Syntax

object.**Cc** [= *value*]

Remarks

The **Cc** property returns the list of addresses that received a copy of the current message. If there is no current message, or the Cc header field is not defined, then this property will return an empty string.

Setting the **Cc** property creates or changes the value of the Cc header field in the message and specifies additional recipients of the message. Multiple addresses may be specified by separating them with a comma. Each address must conform to the standard Internet address format.

Data Type

String

See Also

Bcc Property, From Property, ReplyTo Property, To Property

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.

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 Long
     Dim cchFieldName As Long
     Dim nOffset As Long

     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 = InternetMail1.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
```

Data Type

String

See Also

CertificateExpires Property, CertificateIssued Property, CertificateStatus Property, CertificateSubject Property, Secure Property

CertificateName Property

Gets and sets the common name for the client certificate.

Syntax

object.**CertificateName** [= *name*]

Remarks

This property sets the common name or friendly name of the certificate that should be used to establish the connection with the server. 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. If a certificate name is specified, the certificate must have a private key associated with it, otherwise the connection attempt 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 client 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 may return one of the following values:

Value	Description
mailCertificateNone	No certificate information is available. A secure connection was not established with the server.
mailCertificateValid	The certificate is valid.
mailCertificateNoMatch	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 **ServerName** property is set to an IP address rather than a host name. It is recommended that the client examine the **CertificateSubject** property to determine the domain name of the site that the certificate was issued for.
mailCertificateExpired	The certificate has expired and is no longer valid. The client can examine the **CertificateExpires** property to determine when the certificate expired.
mailCertificateRevoked	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.
mailCertificateUntrusted	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.
mailCertificateInvalid	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

String

See Also

CertificateExpires Property, CertificateIssued Property, CertificateIssuer Property, CertificateSubject Property, Secure Property

CertificateStore Property

Gets and sets the name of the client certificate store or file.

Syntax

object.**CertificateStore** [= *store*]

Remarks

This property sets the name of the certificate store that contains the client certificate that should be used when establishing a secure connection with 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 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

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 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

See Also

CertificateExpires Property, CertificateIssued Property, CertificateIssuer Property, CertificateStatus Property, Secure Property

CertificateUser Property

Gets and sets the user that owns the client certificate.

Syntax

object.**CertificateUser** [= *username*]

Remarks

This property sets the name of the user that owns the client certificate that will be used to establish a secure connection with the server. 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

ContentID Property

Gets and sets the content identifier for the selected message part.

Syntax

object.**ContentID** [= *value*]

Remarks

The **ContentID** property returns the unique content identifier string for the current message part. This multipart header field is not commonly used, and if undefined, will return an empty string. If set, this will change the value of the Content-ID header field in the current message part.

Data Type

String

See Also

ContentLength Property, ContentType Property

ContentLength Property

Returns the size of the data stored in the selected message part.

Syntax

object.**ContentLength**

Remarks

The **ContentLength** property returns the size of the current message part in bytes. This property is read-only, and is automatically updated when the current message part changes.

Data Type

Integer (Int32)

See Also

ContentID Property, ContentType Property

ContentType Property

Gets and sets the content type of the selected message part.

Syntax

object.**ContentType** [= *value*]

Remarks

The **ContentType** property returns the MIME type for the currently selected message part. The type string consists of a primary type and secondary sub-type separated by a slash, followed by one or more optional parameters delimited by semi-colons. For example, this is a common content type for text messages:

```
text/plain; charset=utf-8
```

The **text** designation indicates that this message part contains readable text, and the **plain** sub-type indicates that the text does not contain any special encoding. The optional parameter which follows the content type provides additional information about the content. In this example, it specifies which character set should be used to display the text. The two common character sets used are UTF-8 and US-ASCII.

There are seven predefined, standard content types, each with their own sub-types. The following table lists these types, along with some common sub-types that are found in messages:

Type	Sub-Types	Description
text	plain, richtext, html	Indicates that the message part contains text. This is the most common type found in mail messages; if no content type is explicitly defined, then it is assumed to be plain text.
image	gif, jpeg	Indicates that the message part contains a graphics image.
audio	basic, aiff, wav	Indicates that the message part contains audio data; the basic sub-type is 8-bit PCM encoded audio (commonly found with the .au filename extension).
video	mpeg, avi	Indicates that the message part contains a video clip in the specified format.
application	octet-stream	Indicates that the message part contains application specific data, typically used with the octet-stream sub-type to indicate binary file attachments for executable programs, compressed file archives, etc.
message	rfc822	Indicates that the message part contains a complete RFC 822 compliant message, complete with headers.
multipart	mixed, alternative	Indicates that this is part of a message that contains multiple parts with different content types.

The three most common content types that are used in applications are text/plain for the mail message body, application/octet-stream for binary file attachments and multipart/mixed for messages that contain both text and attached files. For more information about the different content types, refer to the Multipurpose Internet Mail Extensions (MIME) standards document RFC 1521.

Data Type

String

See Also

ContentID Property, ContentLength Property

Date Property

Gets and sets the date for the current message.

Syntax

object.**Date** [= *value*]

Remarks

The **Date** property returns the value of the Date header field in the current message. Setting this property causes the Date field to be updated with the specified value. When setting the date, any one of the following formats may be used:

Format	Example
mm/dd/yy[yy] hh:mm[:ss]	03/01/98 12:00
yy[yy]/mm/dd hh:mm[:ss]	98/03/01 12:00
dd mmm yy[yy] hh:mm[:ss]	01 Mar 1998 12:00:00
mmm dd yy[yy] hh:mm[:ss]	Mar 01 1998 12:00:00

Any extraneous information that may be included in the date string, such as the day of the week, is ignored. In addition to the date and time, the string may also include a time zone specification at the end. If no time zone is specified, the current time zone is used.

When specifying a time zone, the value should either be prefixed by a plus sign (+) to indicate that the time zone is to the east of GMT, or a minus sign (-) to indicates that it's to the west. Four digits follow, with the first two indicating the number of hours east or west of GMT, and the last two digits indicating the number of minutes. Therefore, a value of -0800 means that the time zone is eight hours to the west of GMT, or in other words, the Pacific time zone. Regardless of the format of the string assigned to the property, it always returns the date in the same standard format.

Note that the **Localize** property affects how dates are processed by the control. If enabled, dates are automatically adjusted for the local time zone. By default, localization is disabled.

Data Type

String

See Also

Localize Property

Domain Property

Gets and sets the local domain name.

Syntax

object.**Domain** [= *value*]

Remarks

The **Domain** property specifies the domain name of the local host, and is used to identify the current system when sending messages. If this property is not defined, then the local host name will be used.

Note that explicitly setting the **Domain** property to a value that does not match your local host name may cause some mail servers to reject any messages that you attempt to send.

Data Type

String

Encoding Property

Gets and sets the content encoding for the current message part.

Syntax

object.**Encoding** [= *value*]

Remarks

The **Encoding** property returns a string which specifies the method used for encoding the current message part. Setting this property causes the Content-Transfer-Encoding header value to be updated. The following values are commonly used:

Type	Description
7bit	The default transfer encoding type, which consists of printable ASCII characters.
8bit	Printable ASCII characters, including those characters with the high-bit set (as is common with the ISO Latin-1 character set); this encoding type is not commonly used.
base64	The transfer encoding type commonly used to convert binary data into 7-bit ASCII characters so that it may be transported safely through the mail system.
binary	All characters; binary transfer encoding is rarely used.
quoted-printable	Printable ASCII characters, with non-printable or extended characters represented using their hexadecimal equivalents.
x-uuencode	A transfer encoding type similar in function to the base64 encoding method.

Note that setting this property only updates the Content-Transfer-Encoding header value. To control the actual encoding method used when attaching a file, see the **AttachFile** method.

Data Type

String

See Also

ContentLength Property, ContentType Property, ExtractFile Method, AttachFile Method

From Property

Gets and sets the address of the person who sent the message.

Syntax

object.**From** [= *value*]

Remarks

The **From** property returns the address of the person who sent the message. Setting the property causes the From header field in the current message to be updated with the new value.

Data Type

String

See Also

Bcc Property, Cc Property, ReplyTo Property, To Property

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

IsBlocked Property

Determine 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 can be used to ensure that another blocking operation is not in progress at the time.

If this 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 the control is already performing a blocking operation.

Data Type

Boolean

See Also

LastError Property

IsConnected Property

Determine if the control is connected to a server.

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

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 code.

Syntax

object.**LastError** [= *value*]

Remarks

The **LastError** property can be read to determine the last error that occurred for this instance of the object. If a value is assigned to this property, it must either be zero (to clear the error) or a valid error code.

Data Type

Integer (Int32)

See Also

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

LastMessage Property

Return the number of the last message available on the server.

Syntax

object.**LastMessage**

Remarks

The **LastMessage** property returns the last message available on the server. Note that unlike the **MessageCount** property, this value remains constant even when a message is deleted.

Data Type

Integer (Int32)

See Also

Message Property, MessageCount Property, MessageSize Property

LocalAddress Property

Return the Internet address of the local host.

Syntax

object.**LocalAddress**

Remarks

The **LocalAddress** property returns the Internet address of the local host as a string in dotted notation. If there is an active connection to a server, then the return value will depend on the network interface that was used to establish the connection. If there isn't a connection, then the default address for the local host will be returned.

Data Type

Integer (Int32)

See Also

LocalName Property

Localize Property

Enable or disable message localization.

Syntax

object.**Localize** [= { True | False }]

Remarks

The **Localize** property is used to enable or disable localization features of the object. Currently this only affects the way in which dates are processed. If set to True, the date and time will be adjusted for the local time zone when setting and reading the **Date** property. The default value for this property is False.

Data Type

Boolean

See Also

Date Property

LocalName Property

Return the Internet domain name of the local host.

Syntax

object.**LocalName**

Remarks

The **LocalName** property returns the Internet domain name for the local host. If there is an active connection to a server, then the domain name will depend on the network interface that was used to establish the connection. If there isn't a connection, then the default domain name for the local host will be returned.

Data Type

String

See Also

LocalAddress Property

Mailbox Property

Returns the name of the specified mailbox from a list of mailboxes on the server.

Syntax

object.**Mailbox**(*Index*)

Remarks

The **Mailbox** property array is used to enumerate the available mailboxes on the IMAP server. This is a zero-based array, which means that the index value for the first mailbox is zero. The total number of mailboxes that are available on the server is returned by the **Mailboxes** property.

This property should only be referenced when connected to an IMAP server.

Data Type

String

Example

The following example demonstrates how to use the **Mailbox** property array to populate a listbox that contains the names of the available mailboxes:

```
For nIndex = 0 To InternetMail.Mailboxes - 1
  List1.AddItem InternetMail.Mailbox(nIndex)
Next
```

See Also

Mailboxes Property, MailboxFlags Property, MailboxName Property, MailboxUID Property

Mailboxes Property

Returns the number of mailboxes available on the server.

Syntax

object.**Mailboxes**

Remarks

The **Mailboxes** property returns the total number of mailboxes available to the current account on the server. This property can be used in conjunction with the **Mailbox** property array to enumerate the names of all of the mailboxes which can be selected by the client.

This property should only be referenced when connected to an IMAP server.

Data Type

Integer (Int32)

Example

The following example demonstrates how to use the **Mailboxes** property to populate a listbox that contains the names of the available mailboxes:

```
For nIndex = 0 To InternetMail.Mailboxes - 1
  List1.AddItem InternetMail.Mailbox(nIndex)
Next
```

See Also

Mailbox Property, MailboxFlags Property, MailboxName Property, MailboxUID Property

MailboxFlags Property

Returns one or more flags which identify characteristics of the current mailbox.

Syntax

object.**MailboxFlags**

Remarks

The **MailboxFlags** property returns information about the currently selected mailbox. The value returned is one or more of the following bit flags:

Value	Description	
&H10000	mailFlagNoInferiors	The mailbox does not contain any child mailboxes. In the IMAP protocol, these are referred to as inferior hierarchical mailbox names.
&H40000	mailFlagMarked	The mailbox is marked as being of interest to a client. If this flag is used, it typically means that the mailbox contains messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.
&H80000	mailFlagUnmarked	The mailbox is marked as not being of interest to a client. If this flag is used, it typically means that the mailbox does not contain any messages. An application should not depend on this flag being present for any given mailbox. Some IMAP servers do not support marked or unmarked flags for mailboxes.

Note that this property should only be used when connected to a mail server using the IMAP4 protocol.

Data Type

Integer (Int32)

Example

The following example demonstrates how to check the **MailboxFlags** property to see if the mailbox contains any child mailboxes:

```
If (InternetMail.MailboxFlags And mailFlagNoInferiors) <> 0 Then
    MsgBox "This mailbox does not contain any child mailboxes"
End If
```

See Also

Mailbox Property, Mailboxes Property, MailboxName Property, MailboxUID Property

MailboxName Property

Gets and sets the name of the current mailbox.

Syntax

object.**MailboxName** [= *mailbox*]

Remarks

The **MailboxName** property returns the name of the currently selected mailbox. If no mailbox has been selected by the client, this property will return an empty string. This property is only valid when connected to an IMAP server.

Setting the **MailboxName** property will select a new mailbox in read-write mode. If the client has a different mailbox currently selected, that mailbox will be closed and any messages marked for deletion will be expunged. To prevent deleted messages from being removed from the previous mailbox, call the **UnselectMailbox** method prior to selecting the new mailbox. Setting the **MailboxName** property to an empty string will cause the current mailbox to be unselected, and a new mailbox will not be selected. Before the application can access any messages, it must select a new mailbox.

Selecting a new mailbox will automatically update those properties which provide information about the current mailbox, such as the **MailboxFlags** and **MailboxUID** properties. If an application wishes to update the information for the current mailbox, simply set the **MailboxName** property again with the same mailbox name. Note that this will not cause any messages marked for deletion to be expunged.

The special case-insensitive mailbox name INBOX is used for new messages. Other mailbox names may or may not be case-sensitive depending on the IMAP server's operating system and implementation.

If the mailbox name contains international characters then it is automatically encoded using a modified version of UTF-7 encoding. For example, if a mailbox is named "Håndskrift", the mailbox name created on the server will be the string "H&AOU-ndskrift". The control will automatically decode UTF-7 encoded mailbox names, making the conversion transparent to the application.

Data Type

String

See Also

Mailbox Property, Mailboxes Property, MailboxFlags Property, MailboxUID Property, SelectMailbox Method, UnselectMailbox Method

MailboxSize Property

Return the size of the current mailbox in bytes.

Syntax

object.**MailboxSize**

Remarks

The **MailboxSize** property returns the combined size of all of the available messages in the current mailbox.

Reading this property may require a significant amount of time to calculate the mailbox size if there are a large number of messages in the mailbox. Because it can potentially result in long delays, it is not recommended that an application calculate the mailbox size unless it is absolutely necessary.

Data Type

Integer (Int32)

MailboxUID Property

Returns the unique identifier for the current mailbox.

Syntax

object.**MailboxUID**

Remarks

The **MailboxUID** property returns an integer value which uniquely identifies the mailbox and corresponds to the UIDVALIDITY value returned by the IMAP server. The actual value is determined by the server and should be considered opaque. The protocol specification requires that a mailbox's UID must not change unless the mailbox contents are modified or existing messages in the mailbox have been assigned new UIDs.

An application can use the **MailboxUID** property value in combination with the **MessageUID** property in order to uniquely identify a message on the server. However, the application must take into consideration that the IMAP server can reassign new message UIDs when the mailbox is modified. If the mailbox and message UIDs are being stored on the local system to track what messages have been retrieved from the server, the application must check the UID of the mailbox whenever it is selected. If the mailbox UID has changed, this means that the UIDs for the messages in that mailbox may have changed. The client should resynchronize with the server, and update it's local copy of that mailbox.

Data Type

Integer (Int32)

See Also

Mailbox Property, Mailboxes Property, MailboxFlags Property, MailboxName Property

Mailer Property

Gets and sets the name of the mailer application.

Syntax

object.**Mailer** [= *value*]

Remarks

The **Mailer** property returns the value of the X-Mailer header field in the current message. This is typically used to identify the application that created the message, however it is not required that this be specified. If the header field is not present in the message, this property will return an empty string. Setting this property will change the value of the X-Mailer header. If the property is set to an empty string, the header will be removed from the message.

Data Type

String

See Also

GetHeader Method, SetHeader Method

Message Property

Gets and sets the current message headers and text.

Syntax

object.**Message** [= *value*]

Remarks

The **Message** property returns the current message, including the headers and all message parts, as a string. Setting this property will cause the current message to be cleared and replaced by the new value. The string contents must follow the standard specifications for a message. If the property is set to an empty string, the current message is cleared.

Note that setting the **Message** property will cause the value of the **Bcc** property to be reset to an empty string.

Data Type

String

MessageCount Property

Return the number of messages available on the server.

Syntax

object.**MessageCount**

Remarks

The **MessageCount** property returns the number of messages available to be retrieved from the mail server. When a message is deleted from the mailbox, this value will decrease. To determine the highest valid message number, regardless of any deleted messages, use the **LastMessage** property.

Data Type

Integer (Int32)

See Also

LastMessage Property, Message Property, MessageSize Property

MessageFlags Property

Returns one or more flags which identify characteristics of the current message.

Syntax

object.**MessageFlags** [= *flags*]

Remarks

The **MessageFlags** property returns information about the currently selected message specified by the **MessageIndex** property. The value returned is one or more of the following bit flags:

Value	Value	Description
mailFlagNone	No flags have been set for the current message	
mailFlagAnswered	The message has been answered	
mailFlagDraft	The message is a draft copy and has not been delivered	
mailFlagUrgent	The message has been flagged for urgent or special attention	
mailFlagSeen	The message has been read	
mailFlagRecent	The message has been added to the mailbox recently	
mailFlagDeleted	The message has been marked for deletion	

Setting the **MessageFlags** property changes the flags for the currently selected message. Multiple bit flags can be combined using the bitwise Or operator. An application can test if a flag is set by using the bitwise And operator.

Note that this property should only be used when connected to a mail server using the IMAP4 protocol.

Data Type

Integer (Int32)

Example

The following example demonstrates how to check the **MessageFlags** property to see if the message has been marked for deletion, and if it has, to clear the flag so that it will not be deleted when the mailbox is unselected:

```
If (InternetMail.MessageFlags And mailFlagDeleted) <> 0 Then
    InternetMail.MessageFlags = InternetMail.MessageFlags And Not mailFlagDeleted
End If
```

See Also

MessageCount Property, MessageIndex Property, MessagePart Property, MessageParts Property, MessageSize Property, MessageUID Property, GetMessage Method

MessageID Property

Return a unique identifier for the current message.

Syntax

object.**MessageID**

Remarks

The **MessageID** property returns the value of the Message-ID header field, a string which is assigned by the mail server to uniquely identify the current message.

Data Type

String

See Also

GetHeader Method

MessageIndex Property

Gets and sets the current message number on the server.

Syntax

object.**MessageIndex** [= *value*]

Remarks

The **MessageIndex** property sets or returns the current message number on the server. Message numbers range from 1 through the number of messages available on the server, as returned by the **LastMessage** property. Setting the **MessageIndex** property to an invalid message number will generate an error.

Data Type

Integer (Int32)

See Also

MessageCount Property, MessageFlags Property, MessageSize Property, MessageUID Property

MessagePart Property

Gets and sets the current part in a multipart message.

Syntax

object.**MessagePart** [= *part*]

Remarks

The **MessagePart** property returns the current message part index. All messages have at least one part, which consists of one or more header fields, followed by the body of the message. The default part, part 0, refers to the main message header and body. If the message contains multiple parts (as with a message that contains one or more attached files), the **MessagePart** property can be set to refer to that specific part of the message.

Messages with file attachments typically consist of a message part which describes the contents of the attachment, followed by the attachment itself. For a message with one attached file, there would be a total of three parts. Part 0 would refer to the main message part, which contains the headers such as From, To, Subject, Date and so on. For multipart messages, part 0 typically does not have a message body, since any text is usually created as a separate part (for those messages that do not contain multiple parts, the part 0 body contains the text message). Part 1 would contain the text describing the attachment, and part 2 would contain the attachment itself. If the attached file is binary, then the transfer encoding type would usually be base64.

Data Type

Integer (Int32)

See Also

ContentType Property, ContentLength Property, Encoding Property, MessageParts Property

MessageParts Property

Return the number of parts in the current message.

Syntax

object.**MessageParts**

Remarks

The **MessageParts** property returns the number of parts in the current message. All messages have at least one part, referenced as part 0. Multipart messages will consist of additional parts which may be accessed by setting the **MessageParts** property.

Data Type

Integer (Int32)

See Also

MessagePart Property, AttachFile Method, ExtractFile Method, ExportMessage Method

MessageSize Property

Return the size of the current message in bytes.

Syntax

object.**MessageSize**

Remarks

The **MessageSize** property returns the size of the current message in bytes. The size includes the header and body portion of the message.

Data Type

Integer (Int32)

See Also

Message Property, MessageCount Property

MessageText Property

Return or change the text in the current message part.

Syntax

object.**MessageText** [= *value*]

Remarks

The **MessageText** property returns the body of the current message part. Setting this property replaces the entire message body with the new text. Note that setting the property to an empty string deletes the body of the current message part, but does not delete the message part itself.

Data Type

String

MessageUID Property

Return the UID for the current message on the mail server.

Syntax

object.**MessageUID**

Remarks

The **MessageUID** property returns a string which uniquely identifies the message on the server. The identifier is assigned by the mail server, and retains the same value across multiple client sessions. This value is typically used when the client wants to leave a message on the mail server, but does not wish to retrieve the message contents multiple times. For example, the client can store the UID for each message that it retrieves, but does not delete from the server. The next time that it connects to the mail server, it compares the UID of a message against the stored values. If there is a match, the client knows that the message has already been retrieved, and does not need to do so again.

This property requires that the mail server support the optional UIDL command. If the command is not supported, this property will always return an empty string. Note that the UID for the message comes from the mail server and is not the same as the Message-ID header field in the message itself.

Data Type

String

NameServer Property

Gets and sets the Internet address for a nameserver.

Syntax

object.**NameServer**(*index*) [= *value*]

Remarks

The **NameServer** property array is used to specify one or more nameservers. The address value must be an Internet address in dot notation. The ***index*** specifies which nameserver to set or return a value for. There may be up to four nameservers defined for any single instance of the object.

A nameserver is a computer which converts a domain name, such as microsoft.com, into an IP address which can be used to establish a connection to a server. In addition to mapping domain names, nameservers also can return information about what servers are responsible for handling mail messages for a given domain. These servers are called "mail exchanges" and there may be more than one mail exchange for a domain, each with its own assigned priority. This information is used by the **SendMessage** method to determine the address of the appropriate SMTP server in order to deliver the message to the specified recipient.

If no nameservers are specified, then the default nameservers for the local host will be used. For those systems which use dial-up connections to the Internet, this requires that the system have an active connection established before this object is initialized.

Data Type

String

NewMessages Property

Return the number of new messages available in the current mailbox.

Syntax

object.**NewMessages**

Remarks

The **NewMessages** property returns the number of new, unread messages available to be retrieved from the currently selected mailbox.

This property value is only meaningful when connected to an IMAP server. If the control is connected to a POP3 server, this property will always return the same value as the **MessageCount** property.

Data Type

Integer (Int32)

See Also

MessageCount Property, RecentMessages Property, UnreadMessages Property, CheckMailbox Method

Organization Property

Return or change the name of the organization that created the message.

Syntax

object.**Organization** [= *name*]

Remarks

The **Organization** property returns the name of the organization that sent the current message. Setting this property updates the specified header value. Note that many mailers do not generate an Organization header field, in which case the property value will be an empty string.

Data Type

String

See Also

GetHeader Method, SetHeader Method

Password Property

Gets and sets the password for the current user.

Syntax

object.**Password** [= *value*]

Remarks

The **Password** property specifies the password used to authenticate the user. If the property is not explicitly set, then an application must provide the password to the **Connect** method. Once the connection has been established, this property will be updated with the appropriate value.

If you have assigned a value to the **BearerToken** property, this property will return an empty string. If a value is assigned to this property, the **BearerToken** property will be cleared and standard password authentication will be used when connecting to the mail server.

Data Type

String

See Also

BearerToken Property, UserName Property, Connect Method

Priority Property

Gets and sets the current message priority.

Syntax

object.**Priority** [= *name*]

Remarks

The **Priority** property returns the current priority for the message. Setting this property value causes the X-Priority header to be updated with the specified value.

There is no strict standard for specifying message priority. The convention is to use a number from 1-5, with 1 indicating the highest priority, 3 as normal priority and 5 as the lowest priority. Some mailers follow the number with a space and then text that describes the priority level.

Data Type

String

See Also

GetHeader Method, SetHeader Method

RecentMessages Property

Returns the number of messages which have recently arrived in the mailbox.

Syntax

object.**RecentMessages**

Remarks

The **RecentMessages** property returns the number of messages which have been recently added to the currently selected mailbox. This property is particularly useful when the INBOX mailbox is selected, since it enables the application to check if any new messages have arrived.

This property value is only meaningful when connected to an IMAP server. If the control is connected to a POP3 server, this property will always return the same value as the **MessageCount** property.

Data Type

Integer (Int32)

See Also

Mailbox Property, Mailboxes Property, MailboxFlags Property, MessageCount Property, UnreadMessages Property

Recipient Property

Return the address of a message recipient.

Syntax

object.**Recipient**(*index*)

Remarks

The **Recipient** property array returns the email address of one of the recipients of the current message, as specified by the ***index*** argument. This property enables an application to enumerate all of the recipient addresses for the current message without having to parse the individual **To**, **Cc** and **Bcc** property values. Note that this property array is read-only; to change the recipients for the current message you must set the **To**, **Cc** or **Bcc** properties.

The ***index*** argument specifies which address to return, with a base value of zero up to the number of recipients.

The string returned by the **Recipient** property contains only the actual email address and does not include the name of the recipient or any comments that may have been included with the address. For example, if the **To** property is set to "John Doe <jdoe@company.com>" then the **Recipient** property would return a value of "jdoe@company.com" for that address.

Data Type

String

Example

The following example enumerates all of the recipients for the current message and adds them to a listbox:

```
For nIndex = 0 To InternetMail1.Recipients
    List1.AddItem InternetMail1.Recipient(nIndex)
Next
```

See Also

Bcc Property, Cc Property, Recipients Property, To Property

Recipients Property

Return the number of recipients for the current message.

Syntax

object.**Recipients**

Remarks

The **Recipients** property returns the number of recipients for the current message. This value can be used in conjunction with the **Recipient** property array to enumerate the recipient email addresses for the current message.

Example

The following example enumerates all of the recipients for the current message and adds them to a listbox:

```
For nIndex = 0 To InternetMail1.Recipients
    List1.AddItem InternetMail1.Recipient(nIndex)
Next
```

Data Type

Integer (Int32)

See Also

Recipient Property

RelayServer Property

Gets and sets the host name or address of a relay server.

Syntax

object.**RelayServer** [= *value*]

Remarks

The **RelayServer** property is used to specify an alternate mail server which will deliver messages for the current user.

Normally, when the **SendMessage** method is used, the recipient address is used to determine what mail server is responsible for accepting messages for that user. However, under some circumstances this may not be desirable or even possible. For example, many Internet Service Providers (ISPs) require that customers send all messages through their servers and block any attempt to establish a direct connection with another mail server. Setting the **RelayServer** property to the host name or address of the ISP mail server will cause all messages to be relayed through that server rather than directly to the recipient.

Note that using a mail server as a relay without the permission of the organization or individual who owns that server may violate Acceptable Use Policies and/or Terms of Service agreements with your service provider.

Data Type

String

See Also

ServerName Property, ServerPort Property, RelayPort Property

RelayPort Property

Gets and sets the port number for the specified relay server.

Syntax

object.**RelayPort** [= *value*]

Remarks

The **RelayPort** property defines the port number which is used to establish a connection with the mail server. This property is used in conjunction with the **RelayServer** property to specify an alternate mail server which is responsible for delivering messages for the current user.

If this property is not set, the default SMTP port will be used when connecting to a relay mail server.

Data Type

Integer (Int32)

See Also

RelayServer Property

ReplyTo Property

Gets and sets the address of the person who should receive replies to this message.

Syntax

object.**ReplyTo** [= *value*]

Remarks

The **ReplyTo** property returns the address of the user who should receive replies to the current message. Setting this property updates the Reply-To header field with the specified value.

Data Type

String

See Also

Bcc Property, Cc Property, From Property, To Property

ReturnReceipt Property

Gets and sets the address of the person who should receive a message indicating that the message has been read.

Syntax

object.**ReturnReceipt** [= *value*]

Remarks

The **ReturnReceipt** property returns the address of the person who should receive a message indicating that the current message has been read. Setting this property updates the Disposition-Notification-To header field with the specified value.

Setting the **ReturnReceipt** property does not automatically cause an acknowledgement to be returned to the sender. An application is responsible for checking to make sure the header field contains a valid address and then generating the return receipt message.

Data Type

String

Secure Property

Specify 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 must set this property value to True prior to calling the **Connect** method. Once the connection has been established, the client may retrieve messages from the server as with standard connections.

It is strongly recommended that any application that sets this property to True use error handling to trap any errors that may occur. If the control 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.

Data Type

Boolean

Example

The following example establishes a secure connection to a server and retrieves a message:

```
InternetMail1.ServerType = mailServerPop3
InternetMail1.ServerName = strServerName
InternetMail1.UserName = strUserName
InternetMail1.Password = strPassword
InternetMail1.Secure = True

nError = InternetMail1.Connect()
If nError > 0 Then
    MsgBox "Unable to connect to server " & strServerName, vbExclamation
    Exit Sub
End If

If InternetMail1.CertificateStatus <> mailCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          InternetMail1.Disconnect
          Exit Sub
     End If
End If

nError = InternetMail1.GetMessage(1)
If nError > 0 Then
     InternetMail1.Disconnect
     MsgBox "Unable to retrieve message from server " & strServerName
     Exit Sub
End If

InternetMail1.Disconnect
```

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
stCipherNone	No cipher has been selected. This is not a secure connection with the server.
stCipherRC2	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.
stCipherRC4	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.
stCipherRC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
stCipherDES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher using 56-bit keys.
stCipherDES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively using a 168-bit key length.
stCipherDESX	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.
stCipherAES	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.
stCipherSkipjack	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
stCipherBlowfish	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
mailHashMD5	The MD5 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
mailHashSHA1	The SHA-1 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
mailHashSHA256	The SHA-256 algorithm has been selected.
mailHashSHA384	The SHA-384 algorithm has been selected.
mailHashSHA512	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
stKeyExchangeNone	No key exchange algorithm has been selected. This is not a secure connection with the server.
stKeyExchangeRSA	The RSA public key exchange algorithm has been selected.
stKeyExchangeKEA	The KEA public key exchange algorithm has been selected. This is an improved version of the Diffie-Hellman public key algorithm.
stKeyExchangeDH	The Diffie-Hellman public key exchange algorithm has been selected.
stKeyExchangeECDH	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

ServerName Property

Gets and sets the host name of the current mail server.

Syntax

object.**ServerName** [= *value*]

Remarks

The **ServerName** property returns the name of the mail server that the client is connected to. Setting this property specifies the host name or Internet address for a subsequent connection.

If the **ServerName** property is not explicitly set, then an application must provide the host name or address to the **Connect** method. Once the connection has been established, this property will be updated with the appropriate value. If the server uses a non-standard port number, it can be specified using the **ServerPort** property.

The mail server must support Post Office Protocol v3 (POP3) to retrieve messages. Setting this property does not affect what server is used to deliver messages. See the **RelayServer** and **RelayPort** properties to specify a mail server that is responsible for relaying messages.

Data Type

String

See Also

RelayPort Property, RelayServer Property, ServerPort Property, ServerType Property, Connect Method

ServerPort Property

Gets and sets the port number for the current mail server.

Syntax

object.**ServerPort** [= *value*]

Remarks

The **ServerPort** property returns the port number that was used to establish a connection with the mail server. Setting this property specifies an alternate port number to use for a subsequent connection. A value of zero specifies that the default port should be used for the connection.

The mail server must support Post Office Protocol v3 (POP3) to retrieve messages. Setting this property does not affect what server is used to deliver messages. See the **RelayServer** and **RelayPort** properties to specify a mail server that is responsible for relaying messages.

Data Type

Integer (Int32)

See Also

RelayPort Property, RelayServer Property, ServerName Property, ServerType Property, Connect Method

ServerType Property

Gets and sets the type of mail server the client is connecting to.

Syntax

object.**ServerType** [= *servertype*]

Remarks

The **ServerType** property may be used to specify the mail server type before establishing a connection. If this property is not explicitly set in code, then the control will attempt to automatically determine which protocol should be used based on the value of the **ServerPort** property. By default, the control will attempt to use the POP3 protocol.

This property may return one of the following values:

Value	Description
mailServerDefault	The default server type is determined by the value of the ServerPort property. If that property has not been set, then the control will use the Post Office Protocol (POP3) as the default protocol when establishing the connection.
mailServerPop3	The mail server is using the Post Office Protocol (POP3)
mailServerImap4	The mail server is using the Internet Message Access Protocol (IMAP4)

It is important to note that certain properties and methods in the control are specific to the IMAP4 protocol, such as those that are used to select and enumerate mailboxes. If you are unsure which protocol your mail server supports, contact the system administrator.

Data Type

Integer (Int32)

See Also

Secure Property, ServerName Property, ServerPort Property, Connect Method

Subject Property

Gets and sets the subject of the current message.

Syntax

object.**Subject** [= *value*]

Remarks

The **Subject** property returns the subject of the current message. Setting this property updates the Subject header with the specified value. Note that not all messages have subjects, in which case this property will return an empty string.

Data Type

String

See Also

GetHeader Method, SetHeader Method

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** [= { True | False }]

Remarks

Error handling for 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, methods will not raise an exception if an error occurs. Instead, the application should check the return value of the method and report any errors based on that value. It is the responsibility of the application to interpret the error code and take an appropriate action. This is the default value for the property.

If the **ThrowError** property is set to True, any method which generates an error will cause the component to raise an exception which must be handled or the application will terminate.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of this property. This property only controls how errors are handled when calling methods.

Data Type

Boolean

See Also

LastError Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking network operation is aborted.

Syntax

object.**Timeout** [= *value*]

Remarks

The **Timeout** property controls the amount of time that the component will wait for a network operation to complete before aborting the operation and returning an error. The default value for this property is 20 seconds. It may be required to increase this value if a slow or unreliable network connection is being used.

Data Type

Integer (Int32)

See Also

Connect Method, SendMessage Method

TimeZone Property

Gets and sets the current timezone offset in seconds.

Syntax

object.**TimeZone** [= *value*]

Remarks

The **TimeZone** property returns the current offset from UTC in seconds. Setting the property changes the current timezone offset to the specified value. The value of this property is initially determined by the date and time settings on the local system.

The **TimeZone** property value is used in conjunction with the **Localize** property to control how message date and time localization is handled.

Data Type

Integer (Int32)

Example

The following code enables localization and changes the current timezone to Eastern Standard, which is five hours (18,000 seconds) west of UTC:

```
InternetMail1.Localize = True
InternetMail1.TimeZone = (5 * 60 * 60)
```

See Also

Localize Property

To Property

Gets and sets the recipient of the current message.

Syntax

object.**To** [= *value*]

Remarks

The **To** property returns the list of addresses that received a copy of the current message. If there is no current message, or the To header field is not defined, then this property will return an empty string.

Setting the **To** property creates or changes the value of the To header field in the message and specifies additional recipients of the message. Multiple addresses may be specified by separating them with a comma. Each address must conform to the standard Internet address format.

Data Type

String

See Also

Bcc Property, Cc Property, From Property, ReplyTo Property

Trace Property

Enable or disable network function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the tracing of network function calls and is primarily used as a debugging tool. 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 enabled or disabled for an entire process. This means that once tracing is enabled for a given instance of the object, all of the function calls made by the process will be logged.

If tracing 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 trace file is fairly large. 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.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Return or specify the network function trace output file.

Syntax

object.**TraceFile** [= *value*]

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, then a file named CSTRACE.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 function tracing is enabled per-process, the trace file is shared by all instances of the object being used. 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:

```
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 included in brackets.

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". Those functions which expect Internet 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the current network function tracing flags.

Syntax

object.**TraceFlags** [= *value*]

Remarks

The **TraceFlags** property is used to specify the type of information written to the trace file when network function tracing is enabled. The following values may be used:

Value	Description
mailTraceInfo	All function calls are written to the trace file. This is the default value.
mailTraceError	Only those function calls which fail are recorded in the trace file.
mailTraceWarning	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file.
mailTraceHexDump	All function calls are written to the trace file, plus all the data that is sent or received is logged, in both ASCII and hexadecimal format.

Since network function tracing is enabled per-process, the trace flags are shared by all instances of the object being used.

Warnings are generated when a non-fatal error is returned by a network function. For example, if data is being sent to the server and the error 10035 is returned, a warning is generated 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

UnreadMessages Property

Returns the number of unread messages in the current mailbox.

Syntax

object.**UnreadMessages**

Remarks

The **UnreadMessages** property returns the number of messages which do not have the SEEN flag in the current mailbox. This value is not the same as the number of recent messages in a mailbox, which is based on when the message was stored in the mailbox. To obtain a list of messages that have not been read, use the **SearchMailbox** method with UNSEEN as the search criteria.

It is possible that a message may be flagged as seen if it has been previously accessed by a different mail client. For example, a client may retrieve a message from an INBOX mailbox using the POP3 protocol, which would cause that message to be flagged as seen. This behavior is server dependent, and is most commonly found where the mail server supports both the POP3 and IMAP4 protocols.

Data Type

Integer (Int32)

See Also

MessageCount Property, NewMessages Property, RecentMessages Property, SearchMailbox Method

UserName Property

Gets and sets the current user name.

Syntax

object.**UserName** [= *value*]

Remarks

The **UserName** property specifies the name used to authenticate the user. If the property is not explicitly set, then an application must provide the user name to the **Connect** method. Once the connection has been established, this property will be updated with the appropriate value.

Data Type

String

See Also

BearerToken Property, Password Property, Connect Method

Version Property

Return the current version of the object.

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 Mail Control Methods

Method	Description
AppendMessage	Append text to the current message part
AttachData	Attach the contents of a buffer to the current message
AttachFile	Attach the specified file to the current message
AttachImage	Attach an inline image to the current message
Cancel	Cancel the current operation
ChangePassword	Change the mailbox password for the specified user
CheckMailbox	Create a checkpoint of the currently selected mailbox
ComposeMessage	Compose a new mail message
Connect	Establish a connection with the specified mail server
ClearMessage	Clear the current message
CopyMessage	Copy a message from the current mailbox to another mailbox
CreateMailbox	Creates a new mailbox on the server
CreateMessage	Create a new message
CreatePart	Create a new message part in a multipart message
DeleteHeader	Delete a header field from the current message part
DeleteMailbox	Deletes a mailbox from the server
DeleteMessage	Delete the specified message from the mail server
DeletePart	Delete the specified message part in the current message
Disconnect	Disconnect from the mail server
ExportMessage	Export the current message to a text file
ExtractAllFiles	Extract all file attachments from the current message
ExtractFile	Extract an attached file from the current message
FindAttachment	Search the current message for a file attachment with the specified file name
GetFirstHeader	Return the first header in the current message part
GetHeader	Return the value for the specified header in the current message part
GetNextHeader	Return the next header in the current message part
GetMessage	Retrieve the specified message from the mail server
Idle	Enables mailbox status monitoring for the client session
ImportMessage	Import a new message from the specified text file
Initialize	Initialize the component and load the networking library
ParseAddress	Parse an Internet email address

ParseMessage	Parse the specified string, adding the contents to the current message
Reset	Reset the state of the component
RenameMailbox	Change the name of a mailbox
SearchMailbox	Search the current mailbox for messages that match the specified criteria
SelectMailbox	Selects the specified mailbox for read-write access
SendMessage	Send an email message to one or more recipients
SetHeader	Set the value of a header field in the current message part
StoreMessage	Store the specified message in a file
UndeleteMessage	Removes the deletion flag for the specified message
Uninitialize	Uninitialize the component and unload the networking library
UnselectMailbox	Unselects the current mailbox

AppendMessage Method

Append text to the current message part.

Syntax

object.**AppendMessage**(*MessageText*)

Parameters

MessageText

A string which specifies the text to append.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **AppendMessage** method appends the specified string to the end of the body of text in the current message part. Each line of text contained in the string should be terminated with a carriage-return/linefeed (CRLF) pair, which is recognized as the end-of-line.

See Also

MessagePart Property, ComposeMessage Method, CreatePart Method

AttachData Method

Attach the contents of a buffer to the current message.

Syntax

object.**AttachData**(*Buffer*, [*Length*], [*ContentName*], [*ContentType*], [*Options*])

Parameters

Buffer

A string or byte array which specifies the data to be attached to the message. If an empty string is passed as the argument, no data is attached, but an additional empty message part will be created.

Length

An integer value which specifies the number of bytes of data in the buffer. If this value is omitted, the entire length of of the string or size of the byte array is used.

ContentName

An optional string argument which specifies a name for the data being attached to the message. This typically is used as a file name by the mail client to store the data in. If this parameter is omitted or passed as an empty string then no name is defined and the data is attached as inline content. Note that if a file name is specified with a path, only the base name will be used.

ContentType

An optional string argument which specifies the type of data being attached. The value must be a valid MIME content type. If this parameter is omitted or passed as an empty string, then the buffer will be examined to determine what kind of data it contains. If there is only text characters, then the content type will be specified as "text/plain". If the buffer contains binary data, then the content type will be specified as "application/octet-stream", which is appropriate for any type of data.

Options

An optional integer value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
mailAttachDefault	The data encoding is based on the content type. Text data is not encoded, and binary data is encoded using the standard base64 encoding algorithm. If this argument is omitted, this is the default value used.
mailAttachBase64	The data is always encoded using the standard base64 algorithm, even if the buffer only contains printable text characters.
mailAttachUucode	The data is always encoded using the uuencode algorithm, even if the buffer only contains printable text characters.
mailAttachQuoted	The data is always encoded using the quoted-printable algorithm. This encoding should only be used if the data contains 8-bit text characters.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **AttachData** method attaches the contents of a string or byte buffer to the current message.

Example

The following example demonstrates how to use the **AttachData** method in Visual Basic:

```
Dim hFile As Integer
Dim lpBuffer() As Byte
Dim cbBuffer As Long

' Open a file for binary access and read it into a
' byte array that will be attached to the message

hFile = FreeFile()
Open strDataFile For Binary As hFile
cbBuffer = LOF(hFile)
ReDim lpBuffer(cbBuffer)
Get hFile, , lpBuffer
Close hFile

' Compose a new message and then attach the contents
' of the buffer

InternetMail1.ComposeMessage strFrom, _
                             strTo, _
                             strCc, _
                             strSubject, _
                             strMessage

InternetMail1.AttachData lpBuffer, cbBuffer, strDataFile
```

See Also

Attachment Property, AttachFile Method, ExtractFile Method

AttachFile Method

Attach the specified file to the current message.

Syntax

object.**AttachFile**(*FileName*, [*Options*])

Parameters

FileName

A string that specifies the name of the file to be attached to the message. If the string is empty or the file does not exist, an error will be returned.

Options

An integer value that specifies the type of encoding that will be applied to the attachment. If this argument is not specified, then text files will not be encoded and binary files will be encoded using the standard base64 algorithm.

Value	Description
mailAttachDefault	The file attachment encoding is based on the file content type. Text files are not encoded, and binary files are encoded using the standard base64 algorithm. This is the default option for file attachments.
mailAttachBase64	The file attachment is always encoded using the standard base64 algorithm, even if the attached file is a plain text file.
mailAttachUucode	The file attachment is always encoded using the standard uuencode algorithm, even if the attached file is a plain text file.
mailAttachQuoted	The file attachment is always encoded using quoted-printable encoding. Note that this encoding method is only recommended for text content, typically either as HTML or RTF.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **AttachFile** method attaches the specified file to the current message. If the message already contains one or more file attachments, then it is added to the end of the message. If the message does not contain any attached files, then it is converted to a multipart message and the file is appended to the message.

See Also

ContentType Property, AttachData Method, ExtractFile Method

AttachImage Method

Attach an inline image to the current message.

Syntax

object.**AttachImage**(*FileName*, [*ContentId*])

Parameters

FileName

A string that specifies the name of the file that contains the image which should be attached to the message. If the string is empty or the file does not exist, an error will be returned.

ContentId

An optional string value which specifies the content ID that is associated with the inline image. If this parameter is omitted or is an empty string, a random content ID string will be automatically generated.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **AttachImage** method attaches an inline image to the current message. Unlike a normal file attachment, this method is designed to be used with HTML formatted email messages that display images attached to the message. If the message already contains one or more images or file attachments, then it is added to the end of the message. If the message does not contain any attachments, then it is converted to a multipart message and the image is appended to the message. To attach regular files to the message, use the **AttachFile** method.

See Also

ContentType Property, AttachData Method, ExtractFile Method

Cancel Method

Cancel 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, OnCancel Event

ChangePassword Method

Change the mailbox password for the specified user.

Syntax

object.**ChangePassword**(*UserName*, *OldPassword*, *NewPassword*)

Parameters

UserName

A string that specifies the username for the mailbox.

OldPassword

A string that specifies the current password for the user's mailbox. An error will be returned if this is an empty string.

NewPassword

A string that specifies the new password for the user's mailbox. An error will be returned if this is an empty string, or if the old and new password are the same value.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

The **ChangePassword** method changes the password that will be used to authenticate the specified user. If the ***UserName*** parameter is the same value as the **UserName** property, then successfully changing the password will cause the **Password** property to be updated with the new password

Note that in order to change the user's mailbox password, the server must be running the poppass service on port 106, on the same server. Because passwords are transmitted as clear text (unencrypted), this service is not considered secure and may not be available.

See Also

Password Property, UserName Property

CheckMailbox Method

Create a checkpoint of the currently selected mailbox.

Syntax

object.**CheckMailbox**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CheckMailbox** method requests that the server create a checkpoint of the currently selected mailbox, and updates the current number of new, unread messages available to the client.

When the client requests a checkpoint, the server may perform implementation-dependent housekeeping for that mailbox, such updating the mailbox on disk with the current state of the mailbox in memory. On some systems this command has no effect other than to update the client with the current number of messages in the mailbox.

This function actually sends two IMAP commands. The first is the CHECK command, followed by the NOOP command to poll for any new messages that have arrived. In addition to polling the server for new messages, this command can also be used to ensure the idle timer on the server does not expire and force a disconnect from the client.

This method can only be used when connected to an IMAP server.

See Also

MailboxName Property, MessageCount Property, NewMessages Property, RecentMessages Property

ComposeMessage Method

Compose a new mail message.

Syntax

object.**ComposeMessage**(*From*, *To*, [*Cc*], [*Bcc*], [*Subject*], [*MessageText*], [*MessageHTML*], [*CharacterSet*], [*EncodingType*])

Parameters

From

A string that specifies the sender's email address. Only a single address should be used. After the message has been composed, the **From** property will be updated with this value.

To

A string that specifies one or more recipient email addresses. Multiple email addresses may be specified by separating them with commas. After the message has been composed, the **To** property will be updated with this value.

Cc

An optional string that specifies one or more additional recipient addresses that will receive a copy of the message. If this argument is not specified, then no Cc header field will be created for this message. After the message has been composed, the **Cc** property will be updated with this value.

Bcc

An optional string that specifies one or more additional recipient addresses that will receive a copy of the message. Unlike the ***cc*** argument, these recipients will not be included in the header of the message. If this argument is not specified, then no blind carbon copies of the message will be sent. After the message has been composed, the **Bcc** property will be updated with this value.

Subject

An optional string that specifies the subject for the message. If the argument is not specified, then no Subject header field will be created for this message. After the message has been composed, the **Subject** property will be updated with this value.

MessageText

An optional string that contains the body of the message. Each line of text contained in the string should be terminated with a carriage-return/linefeed (CRLF) pair, which is recognized as the end-of-line. If this parameter is not specified, then the message will have an empty body unless the ***MessageHTML*** parameter has been specified.

MessageHTML

An optional string that contains an alternate HTML formatted message. If the ***MessageText*** parameter has been specified, then a multipart message will be created with both plain text and HTML text as the alternative. This allows mail clients to select which message body they wish to display. If the ***MessageText*** parameter is not specified or is an empty string, then the message will only contain HTML. Although this is supported, it is not recommended because older mail clients may be unable to display the message correctly.

CharacterSet

An optional integer value which specifies the character set for the message text. If this parameter is omitted, the default is for the message to be composed using the standard UTF-8 character set.

EncodingType

An optional integer value which specifies the default encoding for the message. If this parameter is omitted, the message will use standard 8-bit encoding. This parameter may be one of the following values:

Value	Description
mailEncoding7Bit	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the first bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
mailEncoding8Bit	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is used with UTF-8 and other multi-byte character sets.
mailEncodingBinary	Binary encoding is essentially the absence of any encoding performed on the message data, and there is no presumption that the data contains textual information. No character set localization or conversion is performed on binary encoded data. This encoding type is not recommended. Instead, binary data should be encoded using the standard base64 algorithm.
mailEncodingQuoted	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
mailEncodingBase64	Base64 encoding is designed to represent binary data in a form that is not human readable but which can be safely exchanged with servers that only accept 7-bit data. Base64 encoding is typically used with file attachments.
mailEncodingUucode	Uuencoding and uudecoding is a legacy encoding format that was used before the MIME standard was established. This encoding method has largely been replaced by base64 encoding, although it is still commonly used for binary newsgroup postings on USENET. Although this encoding format is supported, it is not officially part of the MIME standard and its use in email messages is discouraged.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ComposeMessage** method will replace the current message if one already exists.

Example

The following example composes a message and sends it to the specified recipients:

```
Dim nError As Long
```

```
nError = InternetMail1.ComposeMessage(comboFrom.Text, _
                                      editTo.Text, _
                                      editCc.Text, _
                                      editBcc.Text, _
                                      editSubject.Text, _
                                      editMessage.Text)
If nError > 0 Then
    MessageBox "Unable to compose message", vbExclamation
    Exit Sub
End If

nError = InternetMail1.SendMessage()

If nError > 0 Then
    MsgBox "Unable to send message", vbExclamation
    Exit Sub
End If
```

See Also

Bcc Property, Cc Property, Encoding Property, From Property, MessageText Property, Subject Property, To Property

Connect Method

Establish a connection with the specified mail server.

Syntax

object.**Connect**([*ServerName*], [*ServerPort*], [*Username*], [*Password*], [*Timeout*], [*Options*])

Parameters

ServerName

A string which specifies the host name or IP address of the mail server.

ServerPort

A number which specifies the port number used to connect to the server. If this argument is not specified, the value of the **ServerType** property will determine the default port number.

UserName

A string which specifies the name of the user used to authenticate access to the server. If this argument is not specified, it defaults to the value of the **UserName** property.

Password

A string which specifies the password used to authenticate the user. If this argument is not specified, it defaults to the value of the **Password** property. If the **BearerToken** property has been assigned a value, this parameter will be ignored and OAuth 2.0 authentication will be used instead of standard password authentication.

Timeout

The number of seconds that the client will wait for a response before failing the operation. If this argument is not specified, the value of the **Timeout** property will be used as the default.

Options

A numeric value which specifies one or more options. If this argument is omitted or a value of zero is specified, a default, standard connection will be established. This argument is constructed by using a bitwise operator with any of the following values:

The settings for *Options* are:

Value	Description
mailOptionImplicitSSL	This option specifies that an implicit TLS session should be established with the mail server and prevents the use of a command which is used to negotiate an explicit TLS connection. This option should only be used if it is required.
mailOptionAPOP	Causes the APOP authentication method to be used when connecting to a POP3 mail server. The default is to use standard password authentication.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Connect** method is used to establish a connection with the specified mail server. This is the first method that must be called prior to the application retrieving mail messages using the **GetMessage**

method. If this method is called when a connection already exists, the current connection will be closed. This has the side-effect of causing any messages which have been marked for deletion to be removed by the mail server.

If the **ServerType** property has not been explicitly set by the application, and a standard port number is specified, then the server type is automatically determined based on the value of the *ServerPort* parameter. If a non-standard port number is specified, then you must set the **ServerType** property to identify whether you are attempting to connect to a POP3 or IMAP4 server.

You should not call the **Connect** method when sending messages using SMTP. This method is only used to establish a connection with a POP3 or IMAP4 server. For more information about sending messages, see the **SendMessage** method and the **RelayServer** and **RelayPort** properties.

Example

The following example connects to a mail server and retrieves each of the mail messages, storing them in a file on the local system:

```
Dim strFileName As String
Dim nMessage As Long, nError As Long

nError = InternetMail1.Connect(strServerName, , strUserName, strPassword)

If nError > 0 Then
    MsgBox "Unable to connect to " & strServerName & vbCrLf & _
           InternetMail1.LastErrorString, vbExclamation
    Exit Sub
End If

If InternetMail1.LastMessage = 0 Then
    MsgBox "The mailbox is currently empty", vbInformation
    InternetMail1.Disconnect
    Exit Sub
End If

For nMessage = 1 To InternetMail1.LastMessage
    strFileName = "c:\temp\msg" & Format(nMessage, "00000") & ".txt"
    nError = InternetMail1.StoreMessage(nMessage, strFileName)
    If nError > 0 Then
        MsgBox "Unable to store message " & nMessage & vbCrLf & _
               InternetMail1.LastErrorString, vbExclamation
        Exit For
    End If
Next

If nError = 0 Then
    MsgBox "Stored " & InternetMail1.LastMessage & " messages", vbInformation
End If

InternetMail1.Disconnect
```

See Also

BearerToken Property, Password Property, Secure Property, ServerName Property, ServerPort Property, ServerType Property, UserName Property, Disconnect Method, GetMessage Method, SendMessage Method

CopyMessage Method

Copy a message from the current mailbox to another mailbox.

Syntax

object.**CopyMessage**(*MessageNumber*, *MailboxName*, [*Options*])

Parameters

MessageNumber

The message identifier which specifies which message is to be copied to the mailbox. This value must be greater than zero and specify a valid message number.

MailboxName

A string which specifies the name of the mailbox that the message will be copied to. The mailbox must already exist, and the client must have the appropriate access rights to modify the mailbox.

Options

An optional parameter reserved for future use. This argument should either be omitted, or always be zero.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CopyMessage** method copies a message from the current mailbox to the specified mailbox. The message is appended to the mailbox, and the message flags and internal date are preserved. If the mailbox does not exist, the function will fail. To create a new mailbox, use the **CreateMailbox** method. A message can be copied within the same mailbox, in which case the server may flag it as a new message.

See Also

CreateMailbox Method, CreateMessage Method, GetMessage Method

ClearMessage Method

Clear the current message.

Syntax

object.**ClearMessage**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ClearMessage** method clears the current message, releasing the memory allocated for the message and any attachments. This will also reset the value of the **Bcc** property back to an empty string.

See Also

ComposeMessage Method

CreateMailbox Method

Creates a new mailbox on the server.

Syntax

object.**CreateMailbox**(*MailboxName*)

Parameters

MailboxName

A string which specifies the name of the new mailbox to be created.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CreateMailbox** method creates a new mailbox on the server. If the mailbox name is suffixed with the server's hierarchy delimiter, this indicates to the server that the client intends to create mailbox names under the specified name in the hierarchy. If superior hierarchical names are specified in the mailbox name, then the server may automatically create them as needed. For example, if the mailbox name "Mail/Office/Projects" is specified and "Mail/Office" does not exist, it may be automatically created by the server.

The special mailbox name INBOX is reserved, and cannot be created. It is recommended that mailbox names only consist of printable ASCII characters, and the special characters "*" and "%" should be avoided.

See Also

CheckMailbox Method, DeleteMailbox Method, SelectMailbox Method

CreateMessage Method

Create a new message.

Syntax

object.**CreateMessage**(*MessageData*, [*MessageFlags*], [*MailboxName*])

Parameters

MessageData

The contents of the message to be created. This may either be specified as a string or as an array of bytes.

MessageFlags

An optional integer value which specifies one or more message flags. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
mailFlagNone	No value.
mailFlagAnswered	The message has been answered.
mailFlagDraft	The message is not completed and is marked as a draft copy.
mailFlagUrgent	The message is flagged for urgent or special attention.
mailFlagSeen	The message has been read.

MailboxName

An optional string argument which specifies the name of the mailbox that the message will be created in. If this argument is omitted, the message will be created in the currently selected mailbox.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CreateMessage** method creates a new message, appending it to the contents of the specified mailbox. This method will cause the current thread to block until the message transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

GetMessage Method, OnProgress Event

CreatePart Method

Create a new message part in a multipart message.

Syntax

object.**CreatePart**([*MessageText*], [*CharacterSet*], [*EncodingType*])

Parameters

MessageText

An optional string that specifies the body of the new message part. Each line of text contained in the string should be terminated with a carriage-return/linefeed (CRLF) pair, which is recognized as the end-of-line. If the parameter is not specified, then the message part will have an empty body.

CharacterSet

An optional integer value that specifies the character set that will be used for this message part. If this parameter is omitted, the message part will use the standard UTF-8 character set. This parameter may be one of the following values:

Value	Description
mailCharsetUSASCII	A character set using US-ASCII which defines 7-bit printable characters with values ranging from 20h to 7Eh. An application that uses this character set has the broadest compatibility with most mail servers (MTAs) because it does not require the server to handle 8-bit characters correctly when the message is delivered. This is the most commonly used character set for plain text email messages in the English language.
mailCharsetISO8859_1	An 8-bit character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin1. The Windows code page for this character set is 28591, however Windows code page 1252 (Windows-1252) is typically used to represent this character set in most applications.
mailCharsetISO8859_2	An 8-bit character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin2. This character set is similar to Windows code page 1250, however the characters are arranged differently.
mailCharsetISO8859_5	An 8-bit character set for Cyrillic languages such as Russian, Bulgarian and Serbian. The Windows code page for this character set is 28595. This character set is not widely used and it is recommended that you use UTF-8 instead.
mailCharsetISO8859_6	An 8-bit character set for Arabic languages. Note that the application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs. This character set is not widely used and it is recommended that you use UTF-8 instead.
mailCharsetISO8859_7	An 8-bit character set for the Greek language. This character set is

	also commonly referred to as Latin/Greek. The Windows code page for this character set is 28597.
mailCharsetISO8859_8	An 8-bit character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English. The Windows code page for this character set is 28598.
mailCharsetISO8859_9	An 8-bit character set for the Turkish language. This character set is also commonly referred to as Latin5. The Windows code page for this character set is 28599.
mailCharsetUTF7	A 7-bit Unicode Transformation Format that uses variable-length character encoding to represent Unicode text as a stream of ASCII characters that are safe to transport between mail servers that only support 7-bit printable characters. It is primarily used as an alternative to UTF-8 which requires that the mail server support 8-bit text or use quoted-printable encoding.
mailCharsetUTF8	An 8-bit Unicode Transformation Format that uses multi-byte character sequences to represent Unicode text. It is backwards compatible with the ASCII character set, however because it uses 8-bit text, it should be encoded using either quoted-printable or base64 encoding to ensure that mail servers that do not support 8-bit characters.

EncodingType

An optional integer value which specifies the encoding for this message part. If this parameter is omitted, the message part will use standard 7-bit encoding. This parameter may be one of the following values:

Value	Description
mailEncoding7Bit	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the first bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
mailEncoding8Bit	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding may be used with multi-byte character sets, although this encoding type is uncommon in email messages.
mailEncodingBinary	Binary encoding is essentially the absence of any encoding performed on the message data, and there is no presumption that the data contains textual information. No character set localization or conversion is performed on binary encoded data. This encoding type is not recommended. Instead, binary data should be encoded using the standard base64 algorithm.
mailEncodingQuoted	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set

	and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
mailEncodingBase64	Base64 encoding is designed to represent binary data in a form that is not human readable but which can be safely exchanged with servers that only accept 7-bit data. Base64 encoding is typically used with file attachments.
mailEncodingUucode	Uuencoding and uudecoding is a legacy encoding format that was used before the MIME standard was established. This encoding method has largely been replaced by base64 encoding, although it is still commonly used for binary newsgroup postings on USENET. Although this encoding format is supported, it is not officially part of the MIME standard and its use in email messages is discouraged.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CreatePart** method creates a new message part. If the current message is a simple RFC822 message, then this method converts it to a MIME multipart message. The current message part will be set to the new part that was just created.

See Also

AttachFile Method, DeletePart Method

DeleteHeader Method

Delete a header field from the current message part.

Syntax

object.**DeleteHeader**(*HeaderField*)

Parameters

HeaderField

A string that specifies the name of the header field to be deleted from the current message part.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **DeleteHeader** method deletes the specified header field value from the current message part.

See Also

ClearMesage Method, GetHeader Method, SetHeader Method

DeleteMessage Method

Delete the specified message from the mail server.

Syntax

object.**DeleteMessage**([*MessageNumber*])

Parameters

MessageNumber

An optional integer value which specifies the message to delete. If this parameter is omitted, the value of the **MessageIndex** property will be used.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **DeleteMessage** method marks the specified message for deletion. If the optional message number is not specified, then the current message is deleted. Once a message has been marked as deleted, any attempt to access it will result in an error.

The message will not actually be removed from the server until the **Disconnect** method is called or the control is unloaded. To prevent messages which have been marked for deletion from actually being removed from the mailbox, call the **Reset** method.

See Also

MessageIndex Property, Disconnect Method, GetHeader Method, GetMessage Method, Reset Method

DeleteMailbox Method

Deletes a mailbox from the server.

Syntax

object.**DeleteMailbox**(*MailboxName*)

Parameters

MailboxName

A string which specifies the name of the mailbox to be deleted.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **DeleteMailbox** method deletes a mailbox from the server. A mailbox cannot be deleted if it contains inferior hierarchical names and has the **mailFlagNoSelect** attribute. On most systems this is the case when the mailbox name references a directory on the server, and that directory contains other subdirectories or mailboxes. To remove the mailbox, you must first delete any child mailboxes that exist.

If the mailbox that is deleted is the currently selected mailbox, it will be automatically unselected and any messages marked for deletion will be expunged before the mailbox is removed. If the delete operation fails, the client will remain in an unselected state until **SelectMailbox** method is called.

The special mailbox name INBOX is reserved, and cannot be deleted.

See Also

MailboxName Property, CheckMailbox Method, CreateMailbox Method, SelectMailbox Method

DeletePart Method

Delete the specified message part in the current message.

Syntax

object.**DeletePart**([*MessagePart*])

Parameters

MessagePart

An optional integer value that specifies the message part to delete from the current message.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **DeletePart** method deletes the specified message part in the current message. If the optional message part is not specified, then the current message part is deleted.

See Also

AttachFile Method, CreatePart Method

Disconnect Method

Disconnect from the mail server.

Syntax

object.**Disconnect**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Disconnect** method causes all messages that have marked for deletion to be removed by the server and the network connection is closed.

See Also

Connect Method

ExportMessage Method

Export the current message to a text file.

Syntax

object.**ExportMessage**(*FileName*, [*Options*])

Parameters

FileName

A string which specifies the name of the file that will contain the message. If the file does not exist, it will be created. If it does exist, it will be overwritten with the contents of the message.

Options

An optional integer value which specifies one or more options. If this argument is omitted, the **Options** property value will be used as the default. The following values may be combined using a bitwise Or operator:

Value	Description	
&H100	mailOptionAllHeaders	All headers, including the Bcc, Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.
&H200	mailOptionKeepOrder	The original order in which the message header fields were set or imported are preserved when the message is exported.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ExportMessage** copies the current message to the specified file. If the file does not exist it will be created, otherwise it will be overwritten with the contents of the message.

See Also

ExtractFile Method, ImportMessage Method

ExtractAllFiles Method

Extract all file attachments from the current message, storing them in the specified directory.

Syntax

object.**ExtractAllFiles**([*Directory*])

Parameters

Directory

An optional string that specifies the name of the directory where the file attachments should be stored. If this parameter is omitted or points to an empty string, the attached files will be stored in the current working directory on the local system.

Return Value

If the method succeeds, the return value is the number of file attachments which were extracted from the current message. If the message does not contain any file attachments, this method will return a value of zero. If the method fails, the return value is -1. To get extended error information, check the value of the **LastError** property.

Remarks

This method will extract all of the files that are attached to the current message and store them in the specified directory. The directory must exist and the current user must have the appropriate permissions to create files there. If a file with the same name as the attachment already exists, it will be overwritten with the contents of the attachment. If the file attachment was encoded using base64 or uuencode, this method will automatically decode the contents of the attachment.

To store a file attachment on the local system using a name that is different than the file name of the attachment, use the **ExtractFile** method.

See Also

Attachment Property, AttachData Method, AttachFile Method, ExtractFile Method

ExtractFile Method

Extract an attached file from the current message.

Syntax

object.**ExtractFile**(*FileName*, [*MessagePart*])

Parameters

FileName

A string which specifies the name of the file that the attachment will be written to. If the file does not exist, it will be created. If the file exists, it will be overwritten.

MessagePart

An optional integer value that specifies the message part that contains the file attachment.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ExtractFile** method writes the contents of a message part, typically a file attachment, to a file on the local system. This method will automatically decode any binary file attachments. If the optional *MessagePart* argument is not specified, the current message part is used. To determine if the current message part contains an attachment and to determine its file name, check the value of the **Attachment** property. An error will be returned if the specified message part does not contain a file attachment.

To search for a file attachment in the current message with a specific file name, use the **FindAttachment** method.

See Also

Attachment Property, ExportMessage Method, ExtractAllFiles Method, FindAttachment Method

FindAttachment Method

Search the current message for a file attachment with the specified file name.

Syntax

object.**FindAttachment**(*FileName*)

Parameters

FileName

A string the specifies the name of the file attachment to search for. This parameter should only specify a base file name; it should not include a file path and cannot be an empty string.

Return Value

If the method succeeds, the return value is the message part number that contains the file attachment. If the message does not contain an attachment that matches the specified file name, the return value is -1. To get extended error information, check the value of the **LastError** property.

Remarks

The **FindAttachment** method will search the current message for a attachment that matches the specified file name. The search is not case-sensitive, however it must match the attachment file name completely. This method will not match partial file names or names that include wildcard characters.

Example

```
nMessagePart = InternetMail1.FindAttachment(strFileName)

If nMessagePart > -1 Then
    InternetMail1.ExtractFile(strFileName, nMessagePart)
End If
```

See Also

Attachment Property, AttachFile Method, ExtractAllFiles Method, ExtractFile Method

GetFirstHeader Method

Return the first header in the current message part.

Syntax

object.**GetFirstHeader**(*HeaderField*, *HeaderValue*)

HeaderField

A string which will contain the name of the first header field when the method returns. This parameter must be passed by reference.

HeaderValue

A string which will contain the value of the first header field when the method returns. This parameter must be passed by reference.

Return Value

A boolean value of True is returned if the method succeeds, otherwise a value of False is returned. For more information about the cause of the failure, check the value of the **LastError** property.

Remarks

The **GetFirstHeader** method allows an application to enumerate all of the headers in the current message. If the current message part does not contain any header fields, this method will return False.

Example

The following example enumerates all of the headers in the main part of the current message and adds them to a listbox:

```
Dim strHeader As String, strValue As String
Dim bResult As Boolean

bResult = InternetMail1.GetFirstHeader(strHeader, strValue)
Do While bResult
    List1.AddItem strHeader & ": " & strValue
    bResult = InternetMail1.GetNextHeader(strHeader, strValue)
Loop
```

See Also

MessagePart Property, GetHeader Method, GetNextHeader Method, SetHeader Method

GetHeader Method

Return the value for the specified header in the current message part.

Syntax

object.**GetHeader**(*HeaderField*, *HeaderValue*)

Parameters

HeaderField

A string variable which will specifies the name of the header field to return the value of. Header field names are not case sensitive.

HeaderValue

A string variable which will contain the value of the specified header field. This parameter must be passed by reference.

Return Value

A boolean value of True is returned if the method succeeds, otherwise a value of False is returned. For more information about the cause of the failure, check the value of the **LastError** property.

Remarks

The **GetHeader** method is used to retrieve the value for a specific header in the current message part. If the header field exists, the method will return True and the ***HeaderValue*** parameter will contain the header value. If the header does not exist, the method will return False.

If there are multiple headers with the same name, the first value will be returned. To enumerate all of the headers in a message, including duplicate header fields, use the **GetFirstHeader** and **GetNextHeader** methods.

See Also

MessagePart Property, GetFirstHeader Method, GetNextHeader Method, SetHeader Method

GetNextHeader Method

Return the next header in the current message part.

Syntax

object.**GetNextHeader**(*HeaderField*, *HeaderValue*)

Parameters

HeaderField

A string which will contain the name of the next header field when the method returns. This parameter must be passed by reference.

HeaderValue

A string which will contain the value of the next header field when the method returns. This parameter must be passed by reference.

Return Value

A boolean value of True is returned if the method succeeds, otherwise a value of False is returned. For more information about the cause of the failure, check the value of the **LastError** property.

Remarks

The **GetNextHeader** method allows an application to enumerate all of the headers in the current message. When all of the headers in the current message part have been returned, this method will return False.

Example

The following example enumerates all of the headers in the main part of the current message and adds them to a listbox:

```
 Dim strHeader As String, strValue As String
Dim bResult As Boolean

bResult = InternetMail1.GetFirstHeader(strHeader, strValue)
Do While bResult
    List1.AddItem strHeader & ": " & strValue
    bResult = InternetMail1.GetNextHeader(strHeader, strValue)
Loop
```

See Also

MessagePart Property, GetFirstHeader Method, GetHeader Method, SetHeader Method

GetMessage Method

Retrieve the specified message from the mail server.

Syntax

object.**GetMessage**([*MessageNumber*])

Parameters

MessageNumber

An optional integer value that specifies the message number.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **GetMessage** method retrieves the specified message from the mail server. This method will cause the current message to be replaced with the new message, and the **MessageIndex** property will be updated with the new message number. If the optional message *MessageNumber* argument is not specified, then the value of the **MessageIndex** property is used instead.

Note that unlike setting the **MessageIndex** property, which only causes the headers for the specified message to be retrieved, the **GetMessage** method downloads the complete message. The **OnProgress** event will fire periodically as the message is retrieved, allowing an application to update its user interface if desired.

Example

The following example connects to a mail server and retrieves the first message:

```
Dim nError As Long

nError = InternetMail1.Connect(strServerName, , strUserName, strPassword)

If nError > 0 Then
    MsgBox "Unable to connect to " & strServerName & vbCrLf & _
           InternetMail1.LastErrorString, vbExclamation
    Exit Sub
End If

If InternetMail1.LastMessage = 0 Then
    MsgBox "The mailbox is currently empty", vbInformation
    InternetMail1.Disconnect
    Exit Sub
End If

nError = InternetMail1.GetMessage(1)

If nError > 0 Then
    MsgBox "Unable to retrieve the message" & vbCrLf & _
           InternetMail1.LastErrorString, vbExclamation
Next

InternetMail1.Disconnect
```

See Also

MessageIndex Property, GetHeader Method, OnProgress Event

Idle Method

Enables mailbox status monitoring for the client session.

Syntax

object.**Idle**([*Options*], [*Timeout*])

Parameters

Options

An optional integer value which specifies how the **Idle** method will function. If this argument is omitted, the method will return immediately to the caller without causing the current thread to block.

Value	Description
mailIdleNoWait	The method should return immediately after idle processing has been enabled. When this option is used, the application may continue to perform other functions while the client session is monitored for status updates sent by the server. The client will continue to monitor status changes until an IMAP command issued or the client disconnects from the server. This is the default option.
mailIdleWait	The method should wait until the server sends a status update, or until the timeout period is reached. The client will stop monitoring status changes when the function returns. If this option is used in a single-threaded application, normal message processing can be impeded, causing the application to appear non-responsive until the timeout period is reached. It is strongly recommended that single-threaded applications with a user interface specify the **mailIdleNoWait** option instead.

Timeout

Specifies the timeout period in seconds to wait for a notification from the server. This parameter is only used when the **mailIdleWait** option has been specified.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Idle** method enables mailbox status monitoring for the client session, allowing the client to receive notifications from the server whenever a new message arrives or a message is expunged from the currently selected mailbox. This is typically used as an alternative to the client periodically polling the server for status information.

Many IMAP servers support the ability to asynchronously send status updates to the client, rather than have the client periodically poll the server. The client enables this feature by calling the **Idle** method and implementing an event handler for the **OnUpdate** event. Typically these events inform the client that a new message has arrived or that a message has been expunged from the mailbox.

The **Idle** method can operate in two modes, based on the options specified by the caller. If the option **mailIdleNoWait** is specified, the method begins monitoring the client session asynchronously and returns control immediately to the caller. If the server sends a update notification to the client, the **OnUpdate** event will fire with information about the status change. If the option **mailIdleWait** is

specified, the method will block waiting for the server to send a notification message to the client. The method will return when either a message is received or the timeout period is exceeded.

Sending an IMAP command to the server will cause the client to stop monitoring the session for status changes. To explicitly stop monitoring the session, use the **Cancel** method.

This method works by sending the IDLE command to the server and starting a worker thread which monitors the connection and looks for untagged responses issued by the server. Events will be generated for EXISTS, EXPUNGE and RECENT messages. Note that some servers may periodically send untagged OK messages to the client, indicating that the connection is still active; these messages are explicitly ignored.

An application should never make an assumption about how a particular server may send update notifications to the client. Servers can be configured to use different intervals at which notifications are sent. For example, a server may send new message notifications immediately, but may periodically notify the client when a message has been expunged. Alternatively, a server may only send notifications at fixed intervals, in which case the client would not be notified of any new messages until the interval period is reached. It is not possible for a client to know what a particular server's update interval is. Applications that require that degree of control should not use the **Idle** method and should poll the server instead.

This method should only be used when connected to an IMAP server. Attempting to use this method when connected to a POP3 server will fail with an error message indicating that the feature is not supported.

See Also

OnUpdate Event

ImportMessage Method

Import a new message from the specified text file.

Syntax

object.**ImportMessage**(*FileName*)

Parameters

FileName

A string that specifies the name of the file to import the message from.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ImportMessage** method replaces the current message with the message contained in the specified text file. Note that calling this method will result in the **Bcc** property value being cleared.

See Also

Bcc Property, AttachFile Method, ExportMessage Method, ExtractFile 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 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 mailClient = CreateObject("SocketTools.InternetMail.11")

nError = mailClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize Method

ParseAddress Method

Parse an Internet email address.

Syntax

object.**ParseAddress**(*Address*)

Parameters

Address

A string value that specifies the address to be parsed.

Return Value

A string that contains the email address or an empty string if the address could not be parsed.

Remarks

The **ParseAddress** method parses a string which contains an email address and returns only the address portion, excluding any comments. An address may contain comments enclosed in parenthesis, or may specify a name along with the address in which case the address is enclosed in angle brackets. For example, consider the following header field value:

```
"User Name" <user@domain.com> (This is a comment)
```

The string "user@domain.com" would be returned if passed the above string, removing the name and any comments.

Note that the **ParseAddress** method will only parse a single address. If multiple addresses are specified, they must be comma delimited and split prior to calling this method.

Example

The following example parses all of the recipient email addresses in the current message, storing them in the *strAddresses* string array.

```
Dim strAddresses() As String, strAddress As String
Dim nIndex As Integer, nAddresses As Integer

nAddresses = 0
strAddresses = Split(InternetMail1.To & "," & _
                     InternetMail1.Cc & "," & _
                     InternetMail1.Bcc, ",")

For nIndex = 0 To UBound(strAddresses)
    If Len(Trim(strAddresses(nIndex))) > 0 Then
        strAddress = InternetMail1.ParseAddress(strAddresses(nIndex))
        If Len(strAddress) > 0 Then
            strAddresses(nAddresses) = strAddress
            nAddresses = nAddresses + 1
        End If
    End If
Next
```

See Also

Recipient Property, Recipients Property, ParseMessage Method

ParseMessage Method

Parse the specified string, adding the contents to the current message.

Syntax

object.**ParseMessage**(*MessageText*)

Parameters

MessageText

A string which specifies the message text to be parsed.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ParseMessage** method parses a string which contains message data, adding it to the current message. This method is useful when the application needs to parse an arbitrary block of text and add it to the current message. If the string contains header fields, the values will be added to the message header. Once the end of the header block is detected, all subsequent text is added to the body of the message.

Note that unlike the **ImportMessage** method, the **ParseMessage** method does not clear the contents of the current message and may be called multiple times. Use the **ClearMessage** method to clear the current message before calling **ParseMessage** if necessary.

Example

The following example demonstrates the use of **ParseMessage** to parse multiple blocks of data from a file. This example effectively does the same thing as calling the **ImportMessage** method:

```
InternetMail1.ClearMessage

hFile = FreeFile()
Open strFileName For Input As hFile
nFileLength = LOF(hFile)

Do While nFileLength > 0
    '
    ' Read the contents of the file in 1K blocks; note that
    ' this is intentionally inefficient to demonstrate
    ' multiple calls to the ParseMessage method.
    '
    cbBuffer = nFileLength: If cbBuffer > 1024 Then cbBuffer = 1024
    nFileLength = nFileLength - cbBuffer
    strBuffer = Input(cbBuffer, hFile)
    '
    ' Parse the string, adding to the current message
    '
    nError = InternetMail1.ParseMessage(strBuffer)
    If nError > 0 Then
        MsgBox InternetMail1.LastErrorString, vbExclamation
        Exit Do
    End If
Loop
```

```
Close hFile
```

See Also

ClearMessage Method, ImportMessage Method

RenameMailbox Method

Change the name of a mailbox.

Syntax

object.**RenameMailbox**(*OldName*, *NewName*)

Parameters

OldName

A string that specifies the name of the mailbox to be renamed on the server. The mailbox must exist on the server, otherwise an error will be returned.

NewName

A string that specifies the new name for the mailbox. An error will be returned if a mailbox with that name already exists.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

If the existing mailbox name contains inferior hierarchical names (mailboxes under the specified mailbox) then those mailboxes will also be renamed. For example, if the mailbox "Mail/Pictures" contains two mailboxes, "Personal" and "Work" and it is renamed to "Mail/Images" then the two mailboxes under it would be automatically renamed to "Mail/Images/Personal" and "Mail/Images/Work".

If the mailbox being renamed is the currently selected mailbox, the current mailbox will be unselected and any messages marked for deletion will be expunged. The new mailbox name will then automatically be re-selected. To prevent deleted messages from being removed from the mailbox prior to being renamed, use the **UnselectMailbox** method to unselect the current mailbox before calling **RenameMailbox**. Note that if the rename operation fails, the client may be left in an unselected state.

It is permitted to rename the special mailbox INBOX. In this case, the messages will be moved from the INBOX mailbox to the new mailbox. If the INBOX mailbox is currently selected, the new mailbox will not automatically be selected. INBOX will remain the selected mailbox.

See Also

Mailbox Property, Mailboxes Property, MailboxName Property, SelectMailbox Method

Reset Method

Reset the state of the component.

Syntax

object.**Reset**

Parameters

None.

Return Value

None.

Remarks

The **Reset** method resets the internal state of the component, releasing any memory allocated for messages and/or network connections. If the application is connected to a mail server, the connection will be terminated. If any messages were marked for pending deletion, those messages will not be deleted.

See Also

Cancel Method

SearchMailbox Method

Search the current mailbox for messages that match the specified criteria.

Syntax

object.**SearchMailbox**(*Criteria*, *Messages*, [*CharacterSet*])

Parameters

Criteria

A string which consists of one or more keywords which are used to define the search criteria. The following keywords are recognized:

Keyword	Description
ANSWERED	Match those messages which have the **mailFlagAnswered** flag set.
BCC *address*	Match those messages which contain the specified address in the BCC header field.
BEFORE *date*	Match those messages which were added to the mailbox prior to the specified date.
BODY *string*	Match those messages where the body contains the specified string.
CC *address*	Match those messages which contain the specified address in the CC header field.
DELETED	Match those messages which have the **mailFlagDeleted** flag set.
DRAFT	Match those messages which have the **mailFlagDraft** flag set.
FLAGGED	Match those messages which have the **mailFlagUrgent** flag set.
FROM *address*	Match those messages which contain the specified address in the FROM header field.
HEADER *field string*	Match those messages which contain the string in the specified header field. If no string is specified, then all messages which contain the header will be matched.
LARGER *size*	Match those messages which are larger than the specified size in bytes.
NEW	Match those messages which have the **mailFlagRecent** flag set, but not the **mailFlagSeen** flag.
OLD	Match those messages which do not have the **mailFlagRecent** flag set.
ON *date*	Match those messages which were added on the specified date.
RECENT	Match those messages which have the **mailFlagRecent** flag set.
SEEN	Match those messages which have the **mailFlagSeen** flag set.
SENTBEFORE *date*	Match those messages whose Date header value is earlier than the specified date.
SENTON *date*	Match those messages whose Date header value is the same as the specified date.
SENTSINCE	Match those messages whose Date header value is later than the

date	specified date.
SINCE *date*	Match those messages added to the mailbox after the specified date.
SMALLER *size*	Match those messages which are smaller than the specified size in bytes.
SUBJECT *string*	Match those messages whose Subject header contains the specified string.
TEXT *string*	Match those messages whose headers or body contains the specified string.
TO *address*	Match those messages which contain the specified address in the TO header field.
UID *sequence*	Match those messages with unique identifiers in the sequence set.
UNANSWERED	Match those messages which do not have the **mailFlagAnswered** flag set.
UNDELETED	Match those messages which do not have the **mailFlagDeleted** flag set.
UNDRAFT	Match those messages which do not have the **mailFlagDraft** flag set.
UNFLAGGED	Match those messages which do not have the **mailFlagUrgent** flag set.
UNSEEN	Match those messages which do not have the **mailFlagSeen** flag set.

Messages

This argument must be passed as an array of integers which will contain the message numbers of those messages which match the search criteria. The size of the array determines the maximum number of matches that will be returned by the method. Note that the array must specify 32-bit integers. In Visual Basic, this means that the array would be typed as Long. In Visual Basic.NET, the array would be typed as Integer.

CharacterSet

An optional string which specifies the character set to use when searching the mailbox. If this argument is omitted, the default UTF-8 character set will be used. Note that not all servers support searches using anything but the default character set.

Return Value

This method will return the number of messages which were found to match the search criteria. If no messages match the criteria, then the return value will be zero. A return value of -1 indicates an error, and the specific error code can be determined by checking the **LastError** property.

Remarks

The **SearchMailbox** method is used to search a mailbox for messages which match a given criteria and return a list of the matching message numbers. The search criteria is composed of one or more search keywords and and optional value to match against. String searches are not case sensitive and partial matches in the message are returned. The message numbers returned by this method are only valid until the mailbox is expunged or another mailbox is selected.

In addition to the listed keywords, the keyword NOT may prefix a keyword to return those messages which do not match the search criteria. For example, "NOT TO user@domain.com" would return those messages which were not addressed to user@domain.com.

If multiple search keywords are specified, the result is the intersection of all those messages which meet the search criteria. For example, a search criteria of "DELETED SINCE 1-Jan-2010" would return

all those messages which are marked for deletion and were added to the mailbox after 1 January 2010.

Those search keywords which expect dates must be specified in format ***dd-mmm-yyyy*** where the month is the three letter abbreviation for the month name. Note that the internal date the message was added to the mailbox is not the same as the value of the Date header field in the message.

If the search keyword expects a string value and the string contains one or more spaces, you need to enclose the search string in quotes as part of the criteria string. For example, in Visual Basic you could use code like this:

```
strCriteria = "SUBJECT " + Chr(34) + "search string" + Chr(34)
```

The quotes around the search string prevents the server from interpreting it as a multiple search criteria to be evaluated. If you are using a search string provided by a user, it is recommended that you always enclose it in quotes to prevent any potential ambiguity in the search. Even if the search string does not contain any spaces, it is always safe to enclose it in quotes.

The UID keyword expects a one or more unique message identifiers. These values may provided as comma separated list, or a range delimited by a colon. For example, "UID 23000:24000" would return all those messages who have UIDs ranging from 23000 through to 24000.

See Also

MailboxName Property, SelectMailbox Method

SelectMailbox Method

Selects the specified mailbox for read-write access.

Syntax

object.**SelectMailbox**(*MailboxName*)

Parameters

MailboxName

A string argument which specifies the name of the mailbox to be selected.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **SelectMailbox** method is used to select a mailbox in read-write mode. If the client has a different mailbox currently selected, that mailbox will be closed and any messages marked for deletion will be expunged. To prevent deleted messages from being removed from the previous mailbox, use the **UnselectMailbox** method prior to selecting the new mailbox.

The special case-insensitive mailbox name INBOX is used for new messages. Other mailbox names may or may not be case-sensitive depending on the IMAP server's operating system and implementation.

See Also

MailboxName Property, CheckMailbox Method, UnselectMailbox Method

SendMessage Method

Send an email message to one or more recipients.

Syntax

object.**SendMessage**([*Sender*], [*Recipient*], [*Message*], [*Options*])

Parameters

Sender

An optional string value that identifies the sender of the message and must be a standard Internet email address. If this argument is omitted, then the address specified by the **From** property will be used.

Recipient

An optional string value that specifies one or more recipients of the message. If this argument is omitted, then the addresses listed in the **Bcc**, **Cc** and **To** properties will be combined to determine the recipients of the message.

Message

An optional string value that contains a complete email message. This must be a properly formatted message that conforms to the standards for Internet email. If this argument is omitted, then the current message is sent.

Options

An optional integer value that specifies one or more options for sending the message. If this argument is omitted, the value of the **Options** property will be used instead. One or more of the following values may be used:

Value	Description
mailOptionImplicitSSL	This option specifies that an implicit TLS session should be established with the mail server and prevents the use of a command which is used to negotiate an explicit TLS connection.
mailOptionAuthLogin	Specifies that the user must be authenticated to the mail server before the message is delivered. This option should only be used if a relay server supports AUTH LOGIN and requires authentication.
&HF0000	mailOptionNotify	Notify the sender of the delivery status of the message, if the server supports delivery status notification. This option is a combination of the *mailNotifySuccess*, *mailNotifyFailure*,

		mailNotifyDelay and *mailReturnHeaders* options.
&H10000	mailNotifySuccess	If the mail server supports delivery status notification, this causes a message to be returned to the sender once it has been successfully delivered.
&H20000	mailNotifyFailure	If the mail server supports delivery status notification, this causes a message to be returned to the sender if it could not be delivered.
&H40000	mailNotifyDelay	If the mail server supports delivery status notification, this causes a message to be returned to the sender if delivery has been delayed.
&H80000	mailReturnHeaders	If the mail server supports delivery status notification, this causes a message to be returned which contains the headers of the message that was sent.
&H100000	mailReturnMessage	If the mail server supports delivery status notification, this causes a message to be returned which contains the complete message that was sent.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **SendMessage** method sends the specified message to one or more recipients. This method can be used in a number of different ways, depending on the arguments specified by the caller.

If the *Message* parameter is specified, but the *sender* and *recipient* properties are omitted, then the message will be parsed and the addresses will be automatically determined by the values of the From, Cc and To header fields. Note that specifying a *message* argument does not change the current message.

For each recipient listed, either as an argument to the method or in the message itself, the **SendMessage** method will determine the appropriate mail exchange server and deliver the message to that user. If the **RelayServer** and **RelayPort** properties are defined, then all messages will be relayed through that specific server, regardless of the recipient address. Note that the **Secure** property and related options only affects connections to relay mail servers. See the **RelayServer** and **RelayPort** properties for additional information.

If a relay server is being used, it may require authentication before accepting any messages for delivery. To enable authentication, specify the **mailOptionAuthLogin** option, either as an argument or by setting the **Options** property. Prior to calling the **SendMessage** method, the **UserName** and

Password properties should be set to the values that will be used to authenticate the session. If the server does not support authentication, or the user name or password is invalid, an error will be returned. Note that authentication is only performed if a relay server is used, otherwise the option is ignored.

See Also

Bcc Property, Cc Property, From Property, RelayPort Property, RelayServer Property, Secure Property, To Property

SetHeader Method

Set the value of a header field in the current message part.

Syntax

object.**SetHeader**(*HeaderField*, *HeaderValue*)

Parameters

HeaderField

A string which specifies the name of the header field to create or modify. If the header field does not exist, then it will be created. If the header field does exist, the value will be overwritten.

HeaderValue

A string which specifies the value of the specified header field.

Return Value

A boolean value of True is returned if the method succeeds, otherwise a value of False is returned. For more information about the cause of the failure, check the value of the **LastError** property.

Remarks

The **SetHeader** method creates or changes the value of the specified header field in the current message part. If the header does not exist, it will be created with the new value. If the header does exist, its current value will be replaced by the new value.

See Also

MessagePart Property, GetFirstHeader Method, GetHeader Method, GetNextHeader Method

StoreMessage Method

Store the specified message in a file.

Syntax

object.**StoreMessage**(*MessageNumber*, *FileName*)

Parameters

MessageNumber

An integer value which specifies the message to store.

FileName

A string value that specifies the name of the file to store the message in.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **StoreMessage** method retrieves the specified message from the mail server and stores it in a file. The ***number*** argument specifies the message to retrieve and the ***filename*** argument specifies the name of the file that the message will be stored in.

For applications which need to store messages on the local system, the **StoreMessage** method is somewhat more efficient than using the **GetMessage** and **ExportMessage** methods to load and store the message. **StoreMessage** does not attempt to analyze the message or change the current message contents.

Example

The following example connects to a mail server and retrieves each of the mail messages, storing them in a file on the local system:

```
Dim strFileName As String
Dim nMessage As Long, nError As Long

nError = InternetMail1.Connect(strServerName, , strUserName, strPassword)

If nError > 0 Then
    MsgBox "Unable to connect to " & strServerName & vbCrLf & _
           InternetMail1.LastErrorString, vbExclamation
    Exit Sub
End If

If InternetMail1.LastMessage = 0 Then
    MsgBox "The mailbox is currently empty", vbInformation
    InternetMail1.Disconnect
    Exit Sub
End If

For nMessage = 1 To InternetMail1.LastMessage
    strFileName = "c:\temp\msg" & Format(nMessage, "00000") & ".txt"
    nError = InternetMail1.StoreMessage(nMessage, strFileName)
    If nError > 0 Then
        MsgBox "Unable to store message " & nMessage & vbCrLf & _
               InternetMail1.LastErrorString, vbExclamation
```

```
        Exit For
    End If
Next

If nError = 0 Then
    MsgBox "Stored " & InternetMail1.LastMessage & " messages", vbInformation
End If

InternetMail1.Disconnect
```

See Also

GetMessage Method, ExportMessage Method

UndeleteMessage Method

Removes the deletion flag for the specified message.

Syntax

object.**UndeleteMessage**(*MessageNumber*)

Parameters

MessageNumber

Number of message to undelete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **UndeleteMessage** method removes the deletion flag for the specified message in the current mailbox. To determine if a message has been marked for deletion, set the **MessageIndex** property to the message number and then check the value of the **MessageFlags** property to determine if the **mailFlagDeleted** bit flag has been set.

This method can only be used when connected to an IMAP server.

See Also

MessageFlags Property, MessageIndex Property, DeleteMessage Method, SelectMailbox 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. 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

UnselectMailbox Method

Unselects the current mailbox.

Syntax

object.**UnselectMailbox**([*Expunge*])

Parameters

Expunge

An optional boolean argument which determines if deleted messages will be expunged from the mailbox. A value of true specifies that messages that have been marked for deletion will be removed from the mailbox. A value of False specifies that no messages will be removed from the mailbox. If this argument is omitted, the default action is to expunge deleted messages from the mailbox.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **UnselectMailbox** method unselects the current mailbox. Once the mailbox has been unselected, no messages in that mailbox can be accessed, and by default any messages which have been marked for deletion are removed.

See Also

MailboxName Property, SelectMailbox Method

Internet Mail Control Events

Event	Description
OnCancel	This event is generated when an operation is canceled
OnDelivered	This event is generated after a message has been delivered
OnError	This event is generated when an error occurs
OnProgress	This event is generated when retrieving or sending messages
OnRecipient	This event is generated before a message is sent
OnTimeout	This event is generated when an operation is canceled
OnUpdate	This event is generated when the server sends a mailbox update notification to the client

OnCancel Event

The **OnCancel** event is generated when an operation is canceled.

Syntax

Private Sub *object*_**OnCancel** ([*Index* **As Integer**])

Remarks

The **OnCancel** event is generated after an operation is canceled by calling the **Cancel** method.

See Also

OnError Event, Cancel Method

OnDelivered Event

The **OnDelivered** event is generated after a message has been delivered.

Syntax

Private Sub *object*_**OnDelivered**([*Index* **As Integer**,] *ByVal Address* **As Variant**, *ByVal MessageSize* **As Variant**)

Remarks

The **OnDelivered** event is generated after a message has been successfully submitted to the mail server for delivery. When used in conjunction with the **OnRecipient** and **OnProgress** events, this event can be used to track the delivery of a message to multiple recipients. If the message was not delivered, either because delivery was canceled in the **OnRecipient** event or because of an error, the **OnDelivered** event will not fire.

The ***Address*** argument is a string which specifies the recipient email address.

The ***MessageSize*** argument is a long integer which specifies the size of the message that was delivered.

Note that even though a message has been successfully delivered to the mail server, it may not actually be delivered to the recipient. The server may accept the message and then subsequently decide to reject or re-route the message based on its own internal configuration. To confirm message delivery to the actual user, use the delivery status notification options and/or set the **ReturnReceipt** property to an address which will be notified when the message has been read.

See Also

OnProgress Event, OnRecipient Event, ReturnReceipt Property

OnError Event

The **OnError** event is generated when an error occurs.

Syntax

Private Sub *object*_**OnError** ([*Index* **As Integer**,] *ByVal Error* **As Variant**, *ByVal Description* **As Variant**)

Remarks

The **OnError** event is generated when an error occurs while the component is performing an operation. 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 component correspond to those returned by the standard Windows Sockets library.

The ***Description*** argument is a string that describes the error. This corresponds to the **LastErrorString** property.

See Also

LastError Property, LastErrorString Property

OnProgress Event

The **OnProgress** event is generated when retrieving or sending messages.

Syntax

Private Sub *object*_**OnProgress** ([*Index* **As Integer**,] *ByVal MessageSize* **As Variant**, *ByVal MessageCopied* **As Variant**, *ByVal Percent* **As Variant**)

Remarks

The **OnProgress** event is generated when a message is being retrieved or sent. This event can be used to update the user interface, such as displaying a progress bar during the transaction. To cancel the current operation, the application can call the **Cancel** method from within this event.

The ***MessageSize*** argument is a long integer which specifies the size of the message in bytes that is currently being sent or received.

The ***MessageCopied*** argument is a long integer which specifies the number of bytes that have been sent or received for the current message.

The ***Percent*** argument is an integer which specifies the completion percentage between a value of 0 and 100.

See Also

Cancel Method

OnRecipient Event

The **OnRecipient** event is generated before a message is sent.

Syntax

Private Sub *object*_**OnRecipient** ([*Index* **As Integer**,] *ByVal Address* **As Variant**, *ByRef Cancel* **As Variant**)

Remarks

The **OnRecipient** event is generated immediately before a message is sent to a recipient. When used in conjunction with the **OnProgress** event, this event can be used to track the delivery of a message to multiple recipients. If an error occurs during the delivery of the message, the **OnError** event will fire.

The ***Address*** argument is a string which specifies the recipient email address.

The ***Cancel*** argument determines whether or not the message delivery is canceled for the specified recipient. Setting this argument to a value of true causes the **SendMessage** method to not deliver the message and continue on to the next recipient. The default value for this argument is False, which indicates that the message should be delivered.

Note that setting the ***Cancel*** argument to True is different from using the **Cancel** method, which would cancel delivery of the message to all subsequent recipients as well as the current recipient specified by the ***Address*** argument.

See Also

Cancel Method, OnProgress Event, OnError Event, SendMessage Method

OnTimeout Event

The **OnTimeout** event is generated when an operation is canceled.

Syntax

Private Sub *object*_**OnTimeout** ([*Index* **As Integer**])

Remarks

The **OnTimeout** event is generated after an operation times out. The amount of time that the component will wait for an operation to complete can be controlled by the **Timeout** property.

See Also

Timeout Property, OnCancel Event

OnUpdate Event

The **OnUpdate** event is generated when the server sends a mailbox update notification to the client.

Syntax

Sub *object*_**OnUpdate** ([*Index* **As Integer**], **ByVal** *UpdateType* **As Variant**, **ByVal** *MessageNumber* **As Variant**)

Remarks

The **OnUpdate** event is generated when the server sends a notification to the client that a new message has been stored in the mailbox, or when a message has been expunged from the mailbox. The arguments to this event are:

UpdateType

An integer value which specifies the type of update notification that has been sent by the server. It may be one of the following values:

Value	Description
mailUpdateUnknown	The server has sent an unrecognized notification message. The value of the ***MessageNumber*** argument is undefined for this type of notification. This does not necessarily reflect an error condition, as some servers may send additional notification messages beyond the standard EXISTS, EXPUNGE and RECENT messages. Most applications should ignore this type of notification.
mailUpdateMessage	The server has sent notification message to the client indicating that a new message has arrived. The ***MessageNumber*** argument will contain the message number for the new message. Typically this update notification occurs shortly after the new message has been stored in the current mailbox.
mailUpdateExpunge	The server has sent a notification message to the client indicating that a message has been removed from the current mailbox. The ***MessageNumber*** argument will contain the message number for the message that has been removed. It is recommended that the application re-examine the mailbox when this notification is received. Typically this notification is only sent periodically by the server, and may not be sent immediately after a message has been expunged from the mailbox.
mailUpdateMailbox	The server has sent notification message to the client indicating that the state of the mailbox has changed. The ***MessageNumber*** argument is not used with this notification. This message is sent periodically by the server and may not be sent immediately after a new message arrives or a message is flagged as unread. It is recommended that the application re-examine the mailbox when this notification is received.

MessageNumber

An integer value which specifies the message number associated with the status change. Note that this argument is not used with the **imapUpdateMailbox** notification and will contain a value of zero.

This event is only generated when the **Idle** method has been used to enable mailbox status monitoring.

See Also

Idle Method

Message Character Sets

Value	Name	Code Page	Description
mimeCharsetUSASCII	us-ascii	A character set which defines 7-bit printable characters with values ranging from 20h to 7Eh. An application that uses this character set has the broadest compatibility with most mail servers (MTAs) because it does not require the server to handle 8-bit characters correctly when the message is delivered.	
mimeCharsetISO8859_1	iso-8859-1	A character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin-1. This character set is similar to Windows code page 1252 (Windows-1252), however there are differences such as the Euro symbol.	
mimeCharsetISO8859_2	iso-8859-2	A character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin-2. This character set is similar to Windows code page 1250, however the characters are arranged differently.	
mimeCharsetISO8859_3	iso-8859-3	A character set for southern European languages such as Maltese and Esperanto. This character set was also used with the Turkish language, but it was superseded by ISO 8859-9 which is the preferred character set for Turkish. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead.	
mimeCharsetISO8859_4	iso-8859-4	A character set for northern European languages such as Latvian, Lithuanian and Greenlandic. This character set is	

		not widely used in mail messages and it is recommended that you use UTF-8 instead.
mimeCharsetISO8859_5	iso-8859-5	A character set for Cyrillic languages such as Russian, Bulgarian and Serbian. This character set was never widely adopted and most mail messages use either KOI8 or UTF-8 encoding.
mimeCharsetISO8859_6	iso-8859-6	A character set for Arabic languages. Note that the application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs.
mimeCharsetISO8859_7	iso-8859-7	A character set for the Greek language. This character set is also commonly referred to as Latin/Greek. This character set is no longer widely used and has largely been replaced with UTF-8 which provides more complete coverage of the Greek alphabet.
mimeCharsetISO8859_8	iso-8859-8	A character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English. In most cases it is recommended that you use UTF-8 instead of this character set.
mimeCharsetISO8859_9	iso-8859-9	A character set for the Turkish language. This character set is also commonly referred to as Latin-5. This character set is nearly identical to ISO 8859-1, except that it replaces certain Icelandic characters with Turkish characters.

mimeCharsetISO8859_10	iso-8859-10	A character set for the Danish, Icelandic, Norwegian and Swedish languages. This character set is also commonly referred to as Latin-6 and is similar to ISO 8859-4.
mimeCharsetISO8859_13	iso-8859-13	A character set for Baltic languages. This character set is also commonly referred to as Latin-7. This character set is similar to ISO 8859-4, except it adds certain Polish characters and does not support Nordic languages.
mimeCharsetISO8859_14	iso-8859-14	A character set for Gaelic languages such as Irish, Manx and Scottish Gaelic. This character set is also commonly referred to as Latin-8. This character set replaced ISO 8859-12 which was never fully implemented.
mimeCharsetISO8859_15	iso-8859-15	A character set for western European languages. This character set is also commonly referred to as Latin-9 and is nearly identical to ISO8859-1 except that it replaces lesser-used symbols with the Euro sign and some letters.
mimeCharsetISO2022_JP	iso-2022-jp	A multi-byte character encoding for Japanese that is widely used with mail messages. This is a 7-bit encoding where all characters start with ASCII and uses escape sequences to switch to the double-byte character sets.
mimeCharsetISO2022_KR	iso-2022-kr	A multi-byte character encoding for Korean which encodes both ASCII and Korean double-byte characters. This is a 7-bit encoding which uses the shift in and shift out control characters to switch to the double-byte character set.
mimeCharsetISO2022_CN	x-cp50227	A multi-byte character encoding for Simplified Chinese which encodes both ASCII and Chinese double-byte characters. This is a 7-bit encoding which uses the shift in

		and shift out control characters to switch to the double-byte character set.
mimeCharsetKOI8R	koi8-r	A character set for Russian using the Cyrillic alphabet. This character set also covers the Bulgarian language. Most mail messages in the Russian language use this character set or UTF-8 instead of ISO 8859-5, which was never widely adopted.
mimeCharsetKOI8U	koi8-u	A character set for Ukrainian using the Cyrillic alphabet. This character set is similar to the KOI8-R character set, but replaces certain symbols with Ukrainian letters. Most mail messages in the Ukrainian language use this character set or UTF-8 instead of ISO 8859-5, which was never widely adopted.
mimeCharsetGB2312	x-cp20936	A multi-byte character encoding which can represent ASCII and simplified Chinese characters. It has been superseded by GB18030, however it remains widely used in China.
mimeCharsetGB18030	gb18030	A Unicode transformation format which can represent all Unicode code points and supports both simplified and traditional Chinese characters. It is backwards compatible with GB2312 and supersedes that character set.
mimeCharsetBIG5	big5	A multi-byte character set that supports both ASCII characters and traditional Chinese characters. It is widely used in Taiwan, Hong Kong and Macau. It is no longer commonly used in China, which has developed GB18030 as a standard encoding. Microsoft's implementation of Big5 on Windows does not support all of the extensions and is missing certain code points.
mimeCharsetUTF7	utf-7	A Unicode transformation format

		that uses variable-length character encoding to represent Unicode text as a stream of ASCII characters that are safe to transport between mail servers that only support 7-bit printable characters. It is primarily used as an alternative to UTF-8 when quoted-printable or base64 encoding is not desired.	
mimeCharsetUTF8	utf-8	A Unicode transformation format that uses multi-byte character sequences to represent Unicode text. It is backwards compatible with the ASCII character set, however because it uses 8-bit text, it is recommended that you use either quoted-printable or base64 encoding to ensure compatibility with mail servers that do not support 8-bit characters.	
mimeCharsetUTF16	utf-16le	N/A	A 16-bit Unicode format that represents each character as a 16-bit value in little endian byte order. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead. UTF-16 characters in big endian byte order are not supported.

Remarks

When composing a new message, it is recommended that you always use UTF-8 as the character set encoding which ensures broad compatibility with most applications. The other character sets are primarily used when parsing messages generated by other applications. Internally, all message headers and text are processed as UTF-8 and returned as Unicode strings.

In addition to the character sets listed above, the control will recognize additional character sets which correspond to specific Windows code pages, as well several variants. These additional character sets are included for compatibility with other applications; they are not defined because they should not be used when composing new messages.

It is important to note that while certain Windows character sets are similar to standard ISO character sets, they are not identical. For example, although the Windows-1252 character set is nearly identical to ISO 8859-1, they are not interchangeable. Some legacy applications make the error of representing Windows ANSI character sets as 8-bit ISO character sets, which can result in errors when converting them to Unicode. This is something to be aware of when encoding and decoding text generated by

older applications. Before the widespread adoption of UTF-8, it was particularly common for legacy Windows mail clients to default to using Windows-1252 for text and label it as using ISO 8859-1.

Although the control supports UTF-16, it is recommended you use UTF-8 instead. Text which uses UTF-16 will always be base64 encoded, and some mail clients may not recognize it as a valid character set. If the message does not specify if big endian or little endian byte order is used, the library will default to little endian. When UTF-16 is used when composing a new message, it will always use little endian byte order.

If you are using this control with Visual Basic 6.0, be aware that the IDE does not provide complete support for Unicode text. Although the control uses Unicode internally, if a header or message body contains characters which cannot be displayed using the current system ANSI code page, the text can appear to be corrupted when examining the string using the debugger. If a message contains text which uses a character set other than the system default, you must use controls which are Unicode aware to display the text, such as the Microsoft InkEdit control. The standard TextBox and other common controls in Visual Basic do not support Unicode.

See Also

ComposeMessage Method, CreatePart Method

Internet Mail Control Error Codes

Value	Constant	Description
10001	mailErrorNotHandleOwner	Handle not owned by the current thread
10002	mailErrorFileNotFound	The specified file or directory does not exist
10003	mailErrorFileNotCreated	The specified file could not be created
10004	mailErrorOperationCanceled	The blocking operation has been canceled
10005	mailErrorInvalidFileType	The specified file is a block or character device, not a regular file
10006	mailErrorInvalidDevice	The specified device or address does not exist
10007	mailErrorTooManyParameters	The maximum number of function parameters has been exceeded
10008	mailErrorInvalidFileName	The specified file name contains invalid characters or is too long
10009	mailErrorInvalidFileHandle	Invalid file handle passed to function
10010	mailErrorFileReadFailed	Unable to read data from the specified file
10011	mailErrorFileWriteFailed	Unable to write data to the specified file
10012	mailErrorOutOfMemory	Out of memory
10013	mailErrorAccessDenied	Access denied
10014	mailErrorInvalidParameter	Invalid argument passed to function
10015	mailErrorClipboardUnavailable	The system clipboard is currently unavailable
10016	mailErrorClipboardEmpty	The system clipboard is empty or does not contain any text data
10017	mailErrorFileEmpty	The specified file does not contain any data
10018	mailErrorFileExists	The specified file already exists
10019	mailErrorEndOfFile	End of file
10020	mailErrorDeviceNotFound	The specified device could not be found
10021	mailErrorDirectoryNotFound	The specified directory could not be found
10022	mailErrorInvalidBuffer	Invalid memory address passed to function
10024	mailErrorNoHandles	No more handles available to this process
10035	mailErrorOperationWouldBlock	The specified operation would block the current thread
10036	mailErrorOperationInProgress	A blocking operation is currently in progress
10037	mailErrorAlreadyInProgress	The specified operation is already in progress
10038	mailErrorInvalidHandle	Invalid handle passed to function
10039	mailErrorInvalidAddress	Invalid network address specified
10040	mailErrorInvalidSize	Datagram is too large to fit in specified buffer
10041	mailErrorInvalidProtocol	Invalid network protocol specified
10042	mailErrorProtocolNotAvailable	The specified network protocol is not available
10043	mailErrorProtocolNotSupported	The specified protocol is not supported
10044	mailErrorSocketNotSupported	The specified socket type is not supported
10045	mailErrorInvalidOption	The specified option is invalid
10046	mailErrorProtocolFamily	The specified protocol family is not supported

10047	mailErrorProtocolAddress	The specified address is invalid for this protocol family
10048	mailErrorAddressInUse	The specified address is in use by another process
10049	mailErrorAddressUnavailable	The specified address cannot be assigned
10050	mailErrorNetworkUnavailable	The networking subsystem is unavailable
10051	mailErrorNetworkUnreachable	The specified network is unreachable
10052	mailErrorNetworkReset	Network dropped connection on reset
10053	mailErrorConnectionAborted	Connection was aborted due to timeout or other failure
10054	mailErrorConnectionReset	Connection was reset by remote network
10055	mailErrorOutOfBuffers	No buffer space is available
10056	mailErrorAlreadyConnected	Connection already established with server
10057	mailErrorNotConnected	No connection established with server
10058	mailErrorConnectionShutdown	Unable to send or receive data after connection shutdown
10060	mailErrorOperationTimeout	The specified operation has timed out
10061	mailErrorConnectionRefused	The connection has been refused by the server
10064	mailErrorHostUnavailable	The specified host is unavailable
10065	mailErrorHostUnreachable	The specified host is unreachable
10067	mailErrorTooManyProcesses	Too many processes are using the networking subsystem
10091	mailErrorNetworkNotReady	Network subsystem is not ready for communication
10092	mailErrorInvalidVersion	This version of the operating system is not supported
10093	mailErrorNetworkNotInitialized	The networking subsystem has not been initialized
10101	mailErrorRemoteShutdown	The server has initiated a graceful shutdown sequence
11001	mailErrorInvalidHostName	The specified hostname is invalid or could not be resolved
11002	mailErrorHostNameNotFound	The specified hostname could not be found
11003	mailErrorHostNameRefused	Unable to resolve hostname, request refused
11004	mailErrorHostNameNotResolved	Unable to resolve hostname, no address for specified host
12001	mailErrorInvalidLicense	The license for this product is invalid
12002	mailErrorProductNotLicensed	This product is not licensed to perform this operation
12003	mailErrorNotImplemented	This function has not been implemented on this platform
12004	mailErrorUnknownLocalHost	Unable to determine local host name
12005	mailErrorInvalidHostAddress	Invalid host address specified
12006	mailErrorInvalidServicePort	Invalid service port number specified
12007	mailErrorInvalidServiceName	Invalid or unknown service name specified
12008	mailErrorInvalidEventId	Invalid event identifier specified
12009	mailErrorOperationNotBlocking	No blocking operation in progress on this socket
12101	mailErrorSecurityNotInitialized	Unable to initialize security interface for this process
12102	mailErrorSecurityContext	Unable to establish security context for this session
12103	mailErrorSecurityCredentials	Unable to open client certificate store or establish client credentials
12104	mailErrorSecurityCertificate	Unable to validate the certificate chain for this session

12105	mailErrorSecurityDecryption	Unable to decrypt data stream
12106	mailErrorSecurityEncryption	Unable to encrypt data stream
12201	mailErrorOperationNotSupported	The specified operation is not supported
12202	mailErrorInvalidProtocolVersion	Invalid application protocol version specified
12203	mailErrorNoServerResponse	No data returned from server
12204	mailErrorInvalidServerResponse	Invalid data returned from server
12205	mailErrorUnexpectedServerResponse	Unexpected response code returned from server
12206	mailErrorServerTransactionFailed	Server transaction failed
12207	mailErrorServiceUnavailable	The service is currently unavailable
12208	mailErrorServiceNotReady	The service is not ready, try again later
12209	mailErrorServerResyncFailed	Unable to resynchronize with server
12210	mailErrorInvalidProxyType	Invalid proxy server type specified
12211	mailErrorProxyRequired	Resource must be accessed through specified proxy
12212	mailErrorInvalidProxyLogin	Unable to login to proxy server using specified credentials
12213	mailErrorProxyResyncFailed	Unable to resynchronize with proxy server
12214	mailErrorInvalidCommand	Invalid command specified
12215	mailErrorInvalidCommandParameter	Invalid command parameter specified
12216	mailErrorInvalidCommandSequence	Invalid command sequence specified
12217	mailErrorCommandNotImplemented	Specified command not implemented on this server
12218	mailErrorCommandNotAuthorized	Specified command not authorized for the current user
12219	mailErrorCommandAborted	Specified command was aborted by the server
12220	mailErrorOptionNotSupported	The specified option is not supported on this server
12221	mailErrorRequestNotCompleted	The current client request has not been completed
12222	mailErrorInvalidUsername	The specified username is invalid
12223	mailErrorInvalidPassword	The specified password is invalid
12224	mailErrorInvalidAccount	The specified account name is invalid
12225	mailErrorAccountRequired	Account name has not been specified
12226	mailErrorInvalidAuthenticationType	Invalid authentication protocol specified
12227	mailErrorAuthenticationRequired	User authentication is required
12228	mailErrorProxyAuthenticationRequired	Proxy authentication required
12229	mailErrorAlreadyAuthenticated	User has already been authenticated
12230	mailErrorAuthenticationFailed	Unable to authenticate the specified user
12251	mailErrorNetworkAdapter	Unable to determine network adapter configuration
12252	mailErrorInvalidRecordType	Invalid record type specified
12253	mailErrorInvalidRecordName	Invalid record name specified
12254	mailErrorInvalidRecordData	Invalid record data specified
12255	mailErrorConnectionOpen	Data connection already established
12256	mailErrorConnectionClosed	Server closed data connection

12257	mailErrorConnectionPassive	Data connection is passive
12258	mailErrorConnectionFailed	Unable to open data connection to server
12259	mailErrorInvalidSecurityLevel	Data connection cannot be opened with this security setting
12260	mailErrorCachedTlsRequired	Data connection requires cached tls session
12261	mailErrorDataReadOnly	Data connection is read-only
12262	mailErrorDataWriteOnly	Data connection is write-only
12263	mailErrorEndOfData	End of data
12264	mailErrorRemoteFileUnavailable	Remote file is unavailable
12265	mailErrorInsufficientStorage	Insufficient storage on server
12266	mailErrorStorageAllocation	File exceeded storage allocation on server
12267	mailErrorDirectoryExists	The specified directory already exists
12268	mailErrorDirectoryEmpty	No files returned by the server for the specified directory
12269	mailErrorEndOfDirectory	End of directory listing
12270	mailErrorUnknownDirectoryFormat	Unknown directory format
12271	mailErrorInvalidResource	Invalid resource name specified
12272	mailErrorResourceRedirected	The specified resource has been redirected
12273	mailErrorResourceRestricted	Access to this resource has been restricted
12274	mailErrorResourceNotModified	The specified resource has not been modified
12275	mailErrorResourceNotFound	The specified resource cannot be found
12276	mailErrorResourceConflict	Request could not be completed due to the current state of the resource
12277	mailErrorResourceRemoved	The specified resource has been permanently removed from this server
12278	mailErrorContentLengthRequired	Request must include the content length
12279	mailErrorRequestPrecondition	Request could not be completed due to server precondition
12280	mailErrorUnsupportedMediaType	Request specified an unsupported media type
12281	mailErrorInvalidContentRange	Content range specified for this resource is invalid
12282	mailErrorInvalidMessagePart	Message is not multipart or an invalid message part was specified
12283	mailErrorInvalidMessageHeader	The specified message header is invalid or has not been defined
12284	mailErrorInvalidMessageBoundary	The multipart message boundary has not been defined
12285	mailErrorNoFileAttachment	The current message part does not contain a file attachment
12286	mailErrorUnknownFileType	The specified file type could not be determined
12287	mailErrorDataNotEncoded	The specified data block could not be encoded
12288	mailErrorDataNotDecoded	The specified data block could not be decoded
12289	mailErrorFileNotEncoded	The specified file could not be encoded
12290	mailErrorFileNotDecoded	The specified file could not be decoded
12291	mailErrorNoMessageText	No message text
12292	mailErrorInvalidCharacterSet	Invalid character set specified
12293	mailErrorInvalidEncodingType	Invalid encoding type specified
12294	mailErrorInvalidMessageNumber	Invalid message number specified

12295	mailErrorNoReturnAddress	No valid return address specified
12296	mailErrorNoValidRecipients	No valid recipients specified
12297	mailErrorInvalidRecipient	The specified recipient address is invalid
12298	mailErrorRelayNotAuthorized	The specified domain is invalid or server will not relay messages
12299	mailErrorMailboxUnavailable	Specified mailbox is currently unavailable
12300	mailErrorMailboxReadonly	The selected mailbox cannot be modified
12301	mailErrorMailboxNotSelected	No mailbox has been selected
12302	mailErrorInvalidMailbox	Specified mailbox is invalid
12303	mailErrorInvalidDomain	The specified domain name is invalid or not recognized
12304	mailErrorInvalidSender	The specified sender address is invalid or not recognized
12305	mailErrorMessageNotDelivered	Message not delivered to any of the specified recipients
12306	mailErrorEndOfMessageData	No more message data available to be read
12307	mailErrorInvalidMessageSize	The specified message size is invalid
12308	mailErrorMessageNotCreated	The message could not be created in the specified mailbox
12309	mailErrorNoMoreMailboxes	No more mailboxes exist on this server
12329	mailErrorInvalidDateFormat	The specified date format is not recognized
12330	mailErrorFeatureNotSupported	The specified feature is not supported on this server
12346	mailErrorNoMessageStore	No message store has been specified
12347	mailErrorMessageStoreChanged	The message store has changed since it was last accessed
12348	mailErrorMessageNotFound	No message was found that matches the specified criteria
12349	mailErrorMessageDeleted	The specified message has been deleted

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	CSWSVX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.InternetServer.11
ClassID	9F5674C0-43F2-4EFF-BB9F-2D9AAD54C187
Threading Model	Apartment
Help File	CSW11HLP.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 compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is

recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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	Enable or disable error handling by the container of the control
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 object

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 returned 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
Text sent and received using a string should be converted using the ANSI code page for the current locale. This is the default encoding type.	
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. It is not recommended that you use this code page unless you know that the remote host is sending text which includes OEM characters.	
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.	
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.	
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.	
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 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

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

IsLocked Property

Determine if the server has been locked using the **SyncLock** method.

This property has been deprecated and may not be included in future versions of the control.

Syntax

object.**IsLocked**

Remarks

The **IsLocked** property returns True if the server has been locked using the **SyncLock** 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 **SyncLock** 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.

Data Type

Boolean

See Also

SyncLock 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

Start Method

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

ServerHandle Property

Return the handle to the socket created to listen for client connections.

Syntax

object.**ServerHandle**

Remarks

The **ServerHandle** read-only property returns the handle of the server socket that was created to listen for client connections. If the server has not been started, a value of -1 is returned. This property can be used in conjunction with direct calls to the Windows Sockets API.

Data Type

Integer (Int32)

See Also

IsListening Property, Start Method, Stop 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

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. For example, in Visual Basic 6.0, the **On Error** statement is used to establish error handling. Note that if an error occurs while a property value is being accessed, an exception will be raised regardless of the value of the **ThrowError** property.

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
Endif
```

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

' 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 socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable (or disable) the tracing of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named CSTRACE.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
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).

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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing 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
swTraceInfo	All function calls are written to the trace file. This is the default value.
swTraceError	Only those function calls which fail are recorded in the trace file.
swTraceWarning	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file.
swTraceHexDump	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.

Since socket function tracing is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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 the error WSAEWOULDBLOCK is returned, a warning is generated since the application simply needs to attempt to write the data at a later time.

Data Type

String

See Also

Trace Property, TraceFile Property

Version Property

Return the current version of the object.

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 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

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 Method

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. Is 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.	
&H1000	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

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 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

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 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 is generated when a client connects to the server
OnCancel	This event is generated when a blocking network operation is canceled
OnConnect	This event is generated when a client connection is established
OnDisconnect	This event is generated when a client connection is terminated
OnError	This event is generated when an error occurs
OnIdle	This event is generated after the last client has disconnected from the server
OnRead	This event is generated when a client has sent data to the server
OnStart	This event is generated when the server has started listening for connections
OnStop	This event is generated when the server has stopped
OnTimeout	This event is generated when a network operation times out
OnWrite	This event is generated 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 is generated 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 is generated 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 is generated 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 is generated 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 is generated 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

Mail Message Control

Compose and parse standard MIME formatted email messages.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	MailMessageCtl.MailMessage
File Name	CSMSGX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.MailMessage.11
ClassID	2EBAA6DD-FFE7-4C6E-AA25-273C399C43E4
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 822, RFC 2045, RFC 2046, RFC 2047, RFC 2048

Overview

The Mail Message ActiveX control provides an interface for composing and processing email messages and newsgroup articles which are structured according to the Multipurpose Internet Mail Extensions (MIME) standard. Using this control, an application can easily create complex messages which include multiple alternative content types, such as plain text and styled HTML text, file attachments and customized headers.

It is not required that the developer understand the complex MIME standard; a single method can be used to create multipart message, complete with a styled HTML text body and support for international character sets. The Mail Message control can be easily integrated with the other mail related components, making it extremely easy to create and process MIME formatted messages.

The control also includes an interface for managing a local message storage file that can be used to store and retrieve multiple messages. Methods are provided to open and create storage files, add, remove and extract messages from storage, and search the stored messages for specific header field values.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a

minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

Mail Message Control Properties

Property	Description
AllHeaders	Returns the complete RFC 822 header values for the current message
AllRecipients	Returns a comma-separated list of all message recipients
Attachment	Gets and sets the name of the current file attachment
Bcc	Gets and sets the blind carbon-copy header field value
Boundary	Returns the boundary string used to separate parts in a multipart message
Cc	Gets and sets the carbon-copy header field value
ContentID	Return the content identifier for the selected message part
ContentLength	Returns the size of the data stored in the selected message part
ContentType	Gets and sets the content type of the selected message part
Date	Gets and sets the date for the current message
Encoding	Gets and sets the content encoding method used for the current message part
From	Gets and sets the address of the user who sent the message
HeaderField	Gets and sets the current header field name
HeaderValue	Gets and sets the value of the specified header field
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error to occur
Localize	Enable or disable message localization
Mailer	Gets and sets the name of the mailer application
Message	Gets and sets the current message headers and body
MessageID	Return the current message identifier
MimeVersion	Gets and sets the MIME version number for the current message
Organization	Gets and sets the name of the organization that originated the message
Part	Gets and sets the current message part
PartCount	Return the number of parts in the current message
Priority	Gets and sets the current message priority
Recipient	Returns the address of a message recipient
Recipients	Returns the number of recipients specified in the current message
ReplyTo	Gets and sets the address of the user who should receive replies to this message
SelLength	Gets and sets the current message body text selection length
SelStart	Gets and sets the starting position of the current message body text selection
SelText	Gets and sets the selected message body text

Sender	Gets and sets the address of the user who originated the message
StoreCount	Gets the number of messages in the current storage file, not including deleted messages
StoreFile	Gets and sets the name of the file used to store messages
StoreIndex	Gets and sets the current message index for the current storage file
StoreSize	Gets the total number of messages in the current storage file, including deleted messages
Subject	Gets and sets the subject of the current message
ThrowError	Enable or disable error handling by the container of the control
TimeZone	Gets and sets the current timezone offset in seconds
Text	Gets and sets the body of the current message part
To	Gets and sets the address of the message recipient
Version	Return the current version of the object

AllHeaders Property

Returns the complete RFC 822 header values for the current message.

Syntax

object.**AllHeaders**

Remarks

The **AllHeaders** property will return all of the RFC 822 header values in a string. This includes the message headers that are most commonly referred to, such as the To, From and Subject headers. Each header and its value are separated by a colon, and terminated with a carriage return and linefeed (CRLF) pair.

The headers and their values returned by this property will not be identical to the header block in the original message. If a header value is split across multiple lines, the text returned by this property will be folded, with the complete header value on a single line of text and removing any extraneous whitespace. If the header value has been encoded by the mail client, this property will return the decoded value, not the original encoded value.

Data Type

String

See Also

HeaderField, HeaderValue, GetHeader, SetHeader

AllRecipients Property

Returns a comma-separated list of all message recipients.

Syntax

object.**AllRecipients**

Remarks

The **AllRecipients** property returns a string value that contains a comma-separated list of all message recipients. To individually enumerate through each of the recipient addresses, you can use the **Recipient** property array and **Recipients** property.

The string returned by this property will include those addresses specified by the **Bcc** property, even though they are not included in the message header.

Data Type

String

See Also

Bcc Property, Cc Property, Recipient Property, Recipients Property, To Property, ComposeMessage Method

Attachment Property

Gets and sets the name of the current file attachment.

Syntax

object.**Attachment** [= *filename*]

Remarks

The **Attachment** property specifies the name of the file attachment in a multipart message. When a new part is selected that contains an attached file, the **Attachment** property is updated to reflect the attached file's name.

This property is used by the attach and extract actions to specify the local file name that will be used. Changing its value does not change the attached file name in the multipart message itself.

Data Type

String

See Also

Boundary Property, FileName Property, AttachFile Method, ExtractFile Method, FindAttachment Method

Bcc Property

Gets and sets the blind carbon-copy header field value.

Syntax

object.**Bcc** [= *addresses*]

Remarks

The **Bcc** property returns the list of addresses that are to receive blind carbon copies of the message. Setting the property creates or modifies the Bcc header field. Multiple addresses can be specified by separating them with commas.

A blind carbon copy is when a copy of a message is delivered to a recipient, but that recipient is not listed in the message headers. Because the other recipients of that same message will not see the address in the headers, they will not know it was delivered to that person. As a result, the Bcc header field is not normally exported when the **ExportMessage** method is called, or when the contents of the message are referenced using the **Message** property. To include the Bcc header in the message, use the **mimeOptionAllHeaders** option. Of course, if this option is specified, the addresses in the Bcc list will no longer be blind to the other recipients.

Data Type

String

See Also

Cc Property, From Property, Message Property, Recipient Property, Recipients Property, ReplyTo Property, Sender Property, To Property, ExportMessage Method

Boundary Property

Returns the boundary string used to separate parts in a multipart message.

Syntax

object.**Boundary**

Remarks

The Boundary property returns the current boundary string being used in a multipart message. When the control is used to create a multipart message, a unique boundary string is created and the Boundary property is updated to reflect it's value.

Data Type

String

See Also

Attachment Property, AttachFile Method, ExtractFile Method

Cc Property

Gets and sets the carbon-copy header field value.

Syntax

object.**Cc** [= *addresses*]

Remarks

The **Cc** property returns the list of addresses that were delivered carbon copies of the message. Setting the property creates or modifies the Cc header field. Multiple addresses can be specified by separating them with commas.

Data Type

String

See Also

Bcc Property, From Property, ReplyTo Property, Sender Property, To Property

ContentID Property

Return the content identifier for the selected message part.

Syntax

object.**ContentID**

Remarks

The **ContentID** property returns the unique content identifier string for the current message part. This multipart header field is not commonly used, and if undefined, will return an empty string.

Data Type

String

See Also

ContentLength Property, ContentType Property

ContentLength Property

Returns the size of the data stored in the selected message part.

Syntax

object.**ContentLength**

Remarks

The **ContentLength** property returns the size of the data (in bytes) stored in the selected message part. This property is read-only, and is updated when the current message part changes.

Data Type

Integer (Int32)

See Also

ContentID Property, ContentType Property

ContentType Property

Gets and sets the content type of the selected message part.

Syntax

object.**ContentType** [= *value*]

Remarks

The **ContentType** property returns the MIME type for the currently selected message part. The type string consists of a primary type and secondary sub-type separated by a slash, followed by one or more optional parameters delimited by semi-colons. For example, this is a common content type for text messages:

text/plain; charset=utf-8

The **text** designation indicates that this message part contains readable text, and the **plain** sub-type indicates that the text does not contain any special encoding. The optional parameter which follows the content type provides additional information about the content. In this example, it specifies which character set should be used to display the text. The two common character sets used are UTF-8 and US-ASCII.

There are seven predefined, standard content types, each with their own sub-types. The following table lists these types, along with some common sub-types that are found in messages:

Type	Sub-Types	Description
text	plain, richtext, html	Indicates that the message part contains text. This is the most common type found in mail messages; if no content type is explicitly defined, then it is assumed to be plain text
image	gif, jpeg	Indicates that the message part contains a graphics image
audio	basic, aiff, wav	Indicates that the message part contains audio data; the basic sub-type is 8-bit PCM encoded audio (commonly found with the .au filename extension)
video	mpeg, avi	Indicates that the message part contains a video clip in the specified format
application	octet-stream, postscript	Indicates that the message part contains application specific data, typically used with the octet-stream sub-type to indicate binary file attachments for executable programs, compressed file archives, etc.
message	rfc822	Indicates that the message part contains a complete RFC 822 compliant message, complete with headers
multipart	mixed, alternative	Indicates that this is part of a mixed message (a message that contains multiple parts of different content types)

The three most common content types that are used in applications are text/plain for the mail message body, application/octet-stream for binary file attachments and multipart/mixed for messages that contain both text and attached files. For more information about the different content types, refer to the Multipurpose Internet Mail Extensions (MIME) standards document RFC 1521.

Data Type

String

See Also

ContentID Property, ContentLength Property

Date Property

Gets and sets the date for the current message.

Syntax

object.**Date** [= *date*]

Remarks

The **Date** property returns the value of the date field in the current message header. Setting this property causes the date field to be updated with the specified value. When setting the date, any one of the following formats may be used:

Format	Example
mm/dd/yy[yy] hh:mm[:ss]	03/01/1998 12:00:00
yy[yy]/mm/dd hh:mm[:ss]	97/03/01 12:00:00
dd mmm yy[yy] hh:mm[:ss]	01 Mar 1998 12:00:00
mmm dd yy[yy] hh:mm[:ss]	Mar 01 1998 12:00:00

Any extraneous information that may be included in the date string, such as the day of the week, is ignored. In addition to the date and time, the string may also include a time zone specification at the end. If no time zone is specified, the current time zone is used.

When specifying a time zone, the value should either be prefixed by a plus sign (+) to indicate that the time zone is to the east of GMT, or a minus sign (-) to indicates that it's to the west. Four digits follow, with the first two indicating the number of hours east or west of GMT, and the last two digits indicating the number of minutes. Therefore, a value of -0800 means that the time zone is eight hours to the west of GMT, or in other words, the Pacific time zone.

Regardless of the format of the string assigned to the property, it always returns the date in the same format (which conforms to the RFC 822 specification). Using the above examples, the date would be returned as "Sat, 01 Mar 1998 12:00:00 -0800".

The **Localize** property affects how dates are processed by the control. If enabled, dates are automatically adjusted for the local time zone. By default, localization is disabled.

Data Type

String

See Also

Localize Property, TimeZone Property

Encoding Property

Gets and sets the content encoding method used for the current message part.

Syntax

object.**Encoding** [= *value*]

Remarks

The **Encoding** property returns the method used for encoding the current message part. Setting this property causes the Content-Transfer-Encoding header value to be updated. The following values are commonly used:

Type	Description
7bit	The default transfer encoding type, which consists of printable ASCII characters.
8bit	Printable ASCII characters, including those characters with the high-bit set (as is common with the ISO Latin-1 character set); this encoding type is not commonly used.
binary	All characters; binary transfer encoding is rarely used.
quoted-printable	Printable ASCII characters, with non-printable or extended characters represented using their hexadecimal equivalents.
base64	The transfer encoding type commonly used to convert binary data into 7-bit ASCII characters so that it may be transported safely through the mail system.
x-uuencode	A transfer encoding type similar in function to the base64 encoding method.

Note that setting this property only updates the Content-Transfer-Encoding header value. To control the actual encoding method used for attachments, specify the encoding method when calling the **AttachFile** method.

Data Type

String

See Also

Attachment Property, ContentLength Property, ContentType Property, ExtractFile Method, AttachFile Method

FileName Property

Gets and sets the name of the file that contains the message to be processed

Syntax

object.**FileName** [= *filename*]

Remarks

The **FileName** property returns the name of the file that contains the message being processed. Setting this property specifies the name of a file that contains a MIME message, and is used in conjunction with the methods used to import and export messages.

Data Type

String

See Also

AttachFile Method, ExportMessage Method, ExtractFile Method, ImportMessage Method

From Property

Gets and sets the address of the user who sent the message.

Syntax

object.**From** [= *address*]

Remarks

The **From** property returns the address of the user who sent the message. Setting the property causes the From header field to be updated with the new value.

Data Type

String

See Also

Bcc Property, Cc Property, ReplyTo Property, Sender Property, To Property

HeaderField Property

Gets and sets the current header field name.

Syntax

object.**HeaderField** [= *header*]

Remarks

The **HeaderField** property returns the name of the current header field. Setting this property causes the control to determine if that header field exists, and if it does, to update the **HeaderValue** property with it's value. This property can be used to obtain the value of any header in the current message part, and in conjunction with the **HeaderValue** property, can be used to create new headers.

Data Type

String

See Also

AllHeaders Property, HeaderValue Property, GetHeader Method, SetHeader Method

HeaderValue Property

Gets and sets the value of the specified header field.

Syntax

object.**HeaderValue** [= *value*]

Remarks

The **HeaderValue** property returns the value of the header specified by the **HeaderField** property. Setting this property updates the specified header value. If the **HeaderField** property refers to a header field that does not exist, then it is created in the current message part.

Data Type

String

See Also

AllHeaders Property, HeaderField Property, GetHeader Method, SetHeader 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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

Localize Property

Enable or disable message localization.

Syntax

object.**Localize** [= { True | False }]

Remarks

The **Localize** property is used to enable or disable localization features of the control. Currently this only affects the way in which dates are processed by the control. If set to True, the control will adjust for the local time zone when setting and reading the **Date** property. The default value for this property is False.

Data Type

Boolean

See Also

Date Property, TimeZone Property

Mailer Property

Gets and sets the name of the mailer application.

Syntax

object.**Mailer** [= *program*]

Remarks

The **Mailer** property returns the value of the X-Mailer field in the current message header. Setting this property causes the field to be updated with the specified value. This is typically used to identify the program which generated the message.

Data Type

String

See Also

HeaderField Property, HeaderValue Property, GetHeader Method, SetHeader Method

Message Property

Gets and sets the current message headers and body.

Syntax

object.**Message** [= *value*]

Remarks

The **Message** property returns the current message, including the headers and all message parts, as a string. Setting this property will cause the current message to be cleared and replaced by the new value. The string contents must follow the standard specifications for a message. If the property is set to an empty string, the current message is cleared.

Data Type

String

See Also

Text Property, ExportMessage Method, ImportMessage Method

MessageID Property

Return the current message identifier.

Syntax

object.**MessageID**

Remarks

The read-only **MessageID** property returns the unique identifier for the current message.

Data Type

String

See Also

HeaderField Property, HeaderValue Property, GetHeader Method

MimeVersion Property

Gets and sets the MIME version number for the current message.

Syntax

object.**MimeVersion** [= *version*]

Remarks

The **MimeVersion** property returns the version number for the current message. Setting this property causes the MIME-Version header value to be changed to the specified value. An empty string causes the MIME version number to be set to the default value of "1.0".

Data Type

String

See Also

HeaderField Property, HeaderValue Property, GetHeader Method, SetHeader Method

Organization Property

Gets and sets the name of the organization that originated the message.

Syntax

object.**Organization** [= *value*]

Remarks

The **Organization** property returns the name of the organization that originated the current message. Setting this property updates the specified header value. Note that many mailers do not generate an Organization header field, in which case the property value will be an empty string.

Data Type

String

See Also

HeaderField Property, HeaderValue Property, GetHeader Method, SetHeader Method

Part Property

Gets and sets the current message part.

Syntax

object.**Part** [= *index*]

Remarks

The **Part** property returns the current message part index. All messages have at least one part, which consists of one or more header fields, followed by the body of the message. The default part, part 0, refers to the main message header and body. If the message contains multiple parts (as with a message that contains one or more attached files), the **Part** property can be set to refer to that specific part of the message.

For example, messages with file attachments typically consist of a message part which describes the contents of the attachment, followed by the attachment itself. For a message with one attached file, there would be a total of three parts. Part 0 would refer to the main message part, which contains the headers such as From, To, Subject, Date and so on. For multipart messages, part 0 typically does not have a message body, since any text is usually created as a separate part (for those messages that do not contain multiple parts, the part 0 body contains the text message). Part 1 would contain the text describing the attachment, and part 2 would contain the attachment itself. If the attached file is binary, then the transfer encoding type would usually be base64.

Data Type

Integer (Int32)

See Also

ContentType Property, ContentLength Property, Encoding Property, PartCount Property

PartCount Property

Return the number of parts in the current message.

Syntax

object.**PartCount**

Remarks

The **PartCount** property returns the number of parts in the current message. All messages have at least one part, referenced as part 0. Multipart messages will consist of additional parts which may be accessed by setting the **Part** property.

Data Type

Integer (Int32)

See Also

Part Property, AttachFile Method, ExportMessage Method, ExtractFile Method

Priority Property

Gets and sets the current message priority.

Syntax

object.**Priority** [= *value*]

Remarks

The **Priority** property returns the current priority for the message. Setting this property value causes the X-Priority header to be updated with the specified value.

There is no standard for specifying message priority. The convention is to use a number from 1-5, with 1 indicating the highest priority, 3 as normal priority and 5 as the lowest priority. Some mailers follow the number with a space and then text that describes the priority level.

Data Type

String

See Also

HeaderField Property, HeaderValue Property, GetHeader Method, SetHeader Method

Recipient Property

Returns the address of a message recipient.

Syntax

object.**Recipient**(*Index*)

Remarks

The **Recipient** property array is used to enumerate the recipient addresses that have been specified in the current message. This includes all of the addresses listed in the To, Cc and Bcc header fields. Only the address itself will be returned, not any comments or extraneous text such as the full name of the recipient. This property array is zero based, meaning that the first index value is zero. The total number of recipients specified in the message can be determined by checking the value of the **Recipients** property.

Data Type

String

Example

The following example demonstrates how to use the **Recipient** property array and the **Recipients** property:

```
' Create a comma separated list of all of the recipient email
' addresses currently specified in the message
Dim strRecipients As String

For nIndex = 0 To MailMessage1.Recipients - 1
  If Len(strRecipients) > 0 Then strRecipients = strRecipients & ", "
  strRecipients = strRecipients & MailMessage1.Recipient(nIndex)
Next
```

See Also

Bcc Property, Cc Property, Recipients Property, To Property, ParseAddress Method

Recipients Property

Returns the number of recipients specified in the current message.

Syntax

object.**Recipients**

Remarks

The **Recipients** property returns the number of recipient addresses that have been specified in the current message. This includes all of the addresses listed in the To, Cc and Bcc header fields. This property can be used in conjunction with the **Recipient** property array to enumerate all of the recipient addresses in the message.

The **AllRecipients** property will return a comma-separated list of all message recipients.

Data Type

Integer (Int32)

Example

The following example demonstrates how to use the **Recipient** property array and the **Recipients** property:

```
' Create a comma separated list of all of the recipient email
' addresses currently specified in the message
Dim strRecipients As String

For nIndex = 0 To MailMessage1.Recipients - 1
  If Len(strRecipients) > 0 Then strRecipients = strRecipients & ", "
  strRecipients = strRecipients & MailMessage1.Recipient(nIndex)
Next
```

See Also

AllRecipients Property, Bcc Property, Cc Property, Recipient Property, To Property

ReplyTo Property

Gets and sets the address of the user who should receive replies to this message.

Syntax

object.**ReplyTo** [= *address*]

Remarks

The **ReplyTo** property returns the address of the user who should receive replies to the current message. Setting this property updates the Reply-To header with the specified value.

Data Type

String

See Also

Bcc Property, Cc Property, From Property, Sender Property, To Property

SelLength Property

Gets and sets the current message body text selection length.

Syntax

object.**SelLength** [= *length*]

Remarks

The **SelLength** property is used to set the length of the text selection in the current message body. When used in conjunction with the **SelStart** property, it can be used to refer to part of a message body.

Data Type

Integer (Int32)

See Also

SelStart Property, SelText Property, Text Property

SelStart Property

Gets and sets the starting position of the current message body text selection.

Syntax

object.**SelStart** [= *offset*]

Remarks

The **SelStart** property specifies a byte offset which is the starting position in the current message body. This property can be used in conjunction with the **SelLength** property to refer to part of a message body.

Data Type

Integer (Int32)

See Also

SelLength Property, SelText Property, Text Property

SelText Property

Gets and sets the selected message body text.

Syntax

object.**SelText** [= *message*]

Remarks

The **SelText** property returns the selected message body text as specified by the **SelStart** and **SelLength** properties. Setting this property replaces text in the message body starting at the byte offset specified by the **SelStart** property.

Data Type

String

See Also

SelLength Property, SelStart Property, Text Property

Sender Property

Gets and sets the address of the user who originated the message.

Syntax

object.**Sender** [= *address*]

Remarks

The **Sender** property returns the address of the user who originated the message. Setting this property updates the X-Sender header with the specified value.

Data Type

String

See Also

Bcc Property, Cc Property, From Property, ReplyTo Property, To Property

StoreCount Property

Returns the number of messages in the current storage file, not including deleted messages.

Syntax

object.**StoreCount**

Remarks

The **StoreCount** property returns the number of messages in the message store. It is important to note that does not count those messages which have been marked for deletion. This means that the value returned by this function will decrease as messages are deleted. To determine the total number of messages, including deleted messages, use the **StoreSize** property.

Data Type

Integer (Int32)

See Also

StoreFile Property, StoreIndex Property, StoreSize Property, DeleteMessage Method, FindMessage Method, OpenStore Method, PurgeStore Method

StoreFile Property

Gets and sets the name of the file used to store messages.

Syntax

object.**StoreFile** [= *filename*]

Remarks

The **StoreFile** property returns the name of the current storage file. Setting this property changes the default filename that is used when opening a new storage file. Note that this property value cannot be changed while a storage file is open; attempting to do so will result in an exception being thrown.

Data Type

String

See Also

StoreCount Property, StoreIndex Property, OpenStore Method

StoreIndex Property

Gets and sets the current message index for the current storage file.

Syntax

object.**StoreIndex** [= *value*]

Remarks

The **StoreIndex** property returns the current message index for the message store. Setting this property changes the current message index. When no storage file has been opened, this property will return a value of zero. After a storage file has been opened, it is changed to a value of one, the first message in the message store. The maximum value for this property is the number of messages in the store, as returned by the **StoreSize** property. Attempting to set this property to a value less than one or greater than the number of messages in the store will result in an exception being thrown.

This property value is updated whenever the **ReadStore** or **ReplaceMessage** methods are used. When the **WriteStore** method is used to store the current message in the message store, this property will be updated to reflect the message index of the newly added message.

Data Type

Integer (Int32)

See Also

StoreCount Property, StoreFile Property, StoreSize Property, FindMessage Method, ReadStore Method, WriteStore Method

StoreSize Property

Returns the total number of messages in the current storage file, including deleted messages.

Syntax

object.**StoreSize**

Remarks

The **StoreSize** property returns the total number of messages in the message store, including those messages that have been deleted. Because the **StoreCount** property value will decrease as messages are deleted, it is recommended that you use this property value when iterating through all of the messages in the message store.

Data Type

Integer (Int32)

See Also

StoreCount Property, StoreFile Property, StoreIndex Property, DeleteMessage Method, FindMessage Method, OpenStore Method, PurgeStore Method

Subject Property

Gets and sets the subject of the current message.

Syntax

object.**Subject** [= *value*]

Remarks

The **Subject** property returns the subject of the current message. Setting this property updates the Subject header with the specified value. Note that not all messages have subjects, in which case this property will be set to an empty string.

Data Type

String

See Also

HeaderField Property, HeaderValue Property, GetHeader Method, SetHeader Method

Text Property

Gets and sets the body of the current message part.

Syntax

object.**Text** [= *value*]

Remarks

The **Text** property returns the body of the current message part. Setting this property replaces the body of the current message part with the new text.

Data Type

String

See Also

SelLength Property, SelStart Property, SelText Property

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
MailMessage1.ThrowError = False
nError = MailMessage1.ImportMessage(strFileName)

If nError > 0 Then
    MsgBox MailMessage1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

MailMessage1.ThrowError = True
MailMessage1.ImportMessage strFileName

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

TimeZone Property

Gets and sets the current timezone offset in seconds.

Syntax

object.**TimeZone** [= *value*]

Remarks

The **TimeZone** property returns the current offset from UTC in seconds. Setting the property changes the current timezone offset to the specified value. The value of this property is initially determined by the date and time settings on the local system.

The **TimeZone** property value is used in conjunction with the **Localize** property to control how message date and time localization is handled.

Data Type

Integer (Int32)

Example

The following code enables localization and changes the current timezone to Eastern Standard, which is five hours (18,000 seconds) west of UTC:

```
MailMessage1.Localize = True
MailMessage1.TimeZone = (5 * 60 * 60)
```

See Also

Localize Property

To Property

Gets and sets the address of the message recipient.

Syntax

object.**To** [= *addresses*]

Remarks

The **To** property returns the address of the message recipient. Setting this property causes the To header to be updated with the specified value. Multiple addresses can be specified by separating them with commas.

Data Type

String

See Also

Bcc Property, Cc Property, From Property, ReplyTo Property, Sender Property

Version Property

Return the current version of the object.

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

Mail Message Control Methods

Method	Description
AppendMessage	Append text to the body of the current message part
AttachData	Attach the contents of a buffer to the current message
AttachFile	Attach the specified file to the current message
AttachImage	Attach an inline image to the current message
ClearMessage	Clear the header and body of the current message
CloseStore	Close the current message storage file
ComposeMessage	Compose a new mail message
CreatePart	Create a new message part in a multipart message
DecodeText	Decode a previously encoded string
DeleteHeader	Delete a header field from the current message part
DeleteMessage	Remove the specified message from the current message store
DeletePart	Delete the specified message part in the current message
EncodeText	Encode a string using base64 or quoted-printable encoding
ExportMessage	Export the current message to a file on the local system
ExtractAllFiles	Extract all file attachments from the current message
ExtractFile	Extract an attachment from the message and store it in a file
FindAttachment	Search the current message for a file attachment with the specified file name
FindMessage	Search for a message in the current message store
GetFirstHeader	Return the first header in the current message part
GetHeader	Return the value for the specified header in the current message part
GetNextHeader	Return the next header in the current message part
ImportMessage	Replace the current message with the contents of a file
Initialize	Initialize the control and validate the runtime license
OpenStore	Open the specified message storage file
ParseAddress	Parse an Internet email address
ParseMessage	Parse the specified string, adding the contents to the current message
PurgeStore	Purge all deleted messages from the current message store
ReadStore	Retrieve a message from the message store, replacing the current message
ReplaceMessage	Replace the specified message in the current message store
Reset	Reset the internal state of the control
SetHeader	Set the value for the specified header in the current message part

Uninitialize	Uninitialize the control and reset it to its default state
WriteStore	Store the current message in the message store

AppendMessage Method

Append text to the body of the current message part.

Syntax

object.**AppendMessage**(*MessageText*)

Parameters

MessageText

A string which specifies the message text to be appended to the current message part.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

If the current message is not a multipart message, it is marked as multipart and the attached file is appended to the message. If the message is already a multipart message, an additional part is created and the attachment is added to the message.

To attach data that is stored in a string or byte array rather than a file, use the **AttachData** method.

See Also

Text Property, ParseMessage Method

AttachData Method

Attach the contents of a buffer to the current message.

Syntax

object.**AttachData**(*Buffer*, [*Length*], [*ContentName*], [*ContentType*], [*Options*])

Parameters

Buffer

A string or byte array which specifies the data to be attached to the message. If an empty string is passed as the argument, no data is attached, but an additional empty message part will be created.

Length

An integer value which specifies the number of bytes of data in the buffer. If this value is omitted, the entire length of of the string or size of the byte array is used.

ContentName

An optional string argument which specifies a name for the data being attached to the message. This typically is used as a file name by the mail client to store the data in. If this parameter is omitted or passed as an empty string then no name is defined and the data is attached as inline content. Note that if a file name is specified with a path, only the base name will be used.

ContentType

An optional string argument which specifies the type of data being attached. The value must be a valid MIME content type. If this parameter is omitted or passed as an empty string, then the buffer will be examined to determine what kind of data it contains. If there is only text characters, then the content type will be specified as "text/plain". If the buffer contains binary data, then the content type will be specified as "application/octet-stream", which is appropriate for any type of data.

Options

An optional integer value which specifies one or more options. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
mimeAttachDefault	The data encoding is based on the content type. Text data is not encoded, and binary data is encoded using the standard base64 encoding algorithm. If this argument is omitted, this is the default value used.
mimeAttachBase64	The data is always encoded using the standard base64 algorithm, even if the buffer only contains printable text characters.
mimeAttachUucode	The data is always encoded using the uuencode algorithm, even if the buffer only contains printable text characters.
mimeAttachQuoted	The data is always encoded using the quoted-printable algorithm. This encoding should only be used if the data contains 8-bit text characters.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **AttachData** method allows an application to attach data to the message as either a file attachment or as inline content. The recipient of the message will see the attached data in the same way that they would see a file attached to the message using the **AttachFile** function.

If the specified message is not a multipart message, it is marked as multipart and the attached file is appended to the message. If the message is already a multipart message, an additional part is created and the attachment is added to the message.

Example

The following example demonstrates how to use the **AttachData** method in Visual Basic:

```
Dim hFile As Integer
Dim lpBuffer() As Byte
Dim cbBuffer As Long

' Open a file for binary access and read it into a
' byte array that will be attached to the message

hFile = FreeFile()
Open strDataFile For Binary As hFile
cbBuffer = LOF(hFile)
ReDim lpBuffer(cbBuffer)
Get hFile, , lpBuffer
Close hFile

' Compose a new message and then attach the contents
' of the buffer

MailMessage1.ComposeMessage strFrom, _
                            strTo, _
                            strCc, _
                            strSubject, _
                            strMessage

MailMessage1.AttachData lpBuffer, cbBuffer, strDataFile
```

See Also

Attachment Property, AttachFile Method, ExtractFile Method

AttachFile Method

Attach the specified file to the current message.

Syntax

object.**AttachFile**(*FileName*, [*Options*])

Parameters

FileName

A string which specifies the name of the file to be attached to the message. If the file is empty or does not exist, an error will be returned by the control.

Options

An optional integer value which specifies how the file will be attached to the message. If this argument is omitted and the file is a text file, it will not be encoded; if the file is a binary file, it will be base64 encoded. To override the default encoding used, specify one of the following options:

Value	Description
mimeAttachDefault	The file attachment encoding is based on the file content type. Text files are not encoded, and binary files are encoded using the standard base64 encoding algorithm. This is the default option for file attachments.
mimeAttachBase64	The file attachment is always encoded using the standard base64 algorithm, even if the attached file is a plain text file.
mimeAttachUucode	The file attachment is always encoded using the uuencode algorithm, even if the attached file is a plain text file.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **AttachFile** method attaches the specified file to the current message. If the message already contains one or more file attachments, then it is added to the end of the message. If the message does not contain any attached files, then it is converted to a multipart message and the file is appended to the message.

The **AttachImage** method can be used to attach an inline image to the message.

See Also

Boundary Property, ContentType Property, AttachImage Method, ExtractFile Method

AttachImage Method

Attach an inline image to the current message.

Syntax

object.**AttachImage**(*FileName*, [*ContentId*])

Parameters

FileName

A string which specifies the name of the image file to be attached to the message. If the file is empty or does not exist, an error will be returned by the control.

ContentId

An optional string which specifies the content ID that is associated with the inline image. If this parameter is omitted or is an empty string, a random content ID string will be automatically generated.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **AttachImage** method attaches an inline image to the current message. Unlike a normal file attachment, this method is designed to be used with HTML formatted email messages that display images attached to the message. If the message already contains one or more images or file attachments, then it is added to the end of the message. If the message does not contain any attachments, then it is converted to a multipart message and the image is appended to the message.

The **AttachFile** method can be used to add standard file attachments to the message.

See Also

Boundary Property, ContentType Property, AttachFile Method, ExtractFile Method

ClearMessage Method

Clear the header and body of the current message.

Syntax

object.**ClearMessage**

Parameters

None.

Return Value

A value of zero is returned if the action was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ClearMessage** method clears the current message, releasing the memory allocated for the message and any attachments.

See Also

SelLength Property, SelStart Property, SelText Property, Text Property

CloseStore Method

Close the current message storage file.

Syntax

object.**CloseStore**

Parameters

None.

Return Value

A value of zero is returned if the action was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CloseStore** method closes the storage file that was previously opened, releasing all of the memory allocated for the message store and purging all deleted messages. This method must be called when the application has finished accessing the messages in the message store.

When the control instance is released by its container, the storage file will automatically be closed. To prevent deleted messages from being removed from the message store, use the **Reset** method.

See Also

StoreFile Property, OpenStore Method, PurgeStore Method, ReadStore Method, WriteStore Method

ComposeMessage Method

Compose a new mail message.

Syntax

object.**ComposeMessage**(*From*, *To*, [*Cc*], [*Bcc*], [*Subject*], [*MessageText*], [*MessageHTML*], [*CharacterSet*], [*EncodingType*])

Parameters

From

A string argument which specifies the sender's email address. Only a single address should be specified. After the message has been composed, the **From** property will be updated with this value.

To

A string argument which specifies one or more recipient email addresses. Multiple email addresses may be specified by separating them with commas. After the message has been composed, the **To** property will be updated with this value.

Cc

An optional string which specifies one or more additional recipient addresses that will receive a copy of the message. If this argument is not specified, then no Cc header field will be created for this message. After the message has been composed, the **Cc** property will be updated with this value.

Bcc

An optional string which specifies one or more additional recipient addresses that will receive a "blind" copy of the message. If this argument is not specified, then no Bcc header field will be created for this message. After the message has been composed, the **Bcc** property will be updated with this value. Note that the Bcc header field is not normally included in the header when the message is exported.

Subject

An optional string argument which specifies the subject for the message. If the argument is not specified, then no Subject header field will be created for this message. After the message has been composed, the **Subject** property will be updated with this value.

MessageText

An optional string argument which specifies the body of the message. Each line of text contained in the string should be terminated with a carriage-return/linefeed (CRLF) pair, which is recognized as the end-of-line. If the argument is not specified, then the message will have an empty body unless the *MessageHTML* argument has been specified.

MessageHTML

An optional argument which specifies an alternate HTML formatted message. If the *MessageText* argument has been specified, then a multipart message will be created with both plain text and HTML text as the alternative. This allows mail clients to select which message body they wish to display. If the *MessageText* argument is not specified or is an empty string, then the message will only contain HTML. Although this is supported, it is not recommended because older mail clients may be unable to display the message correctly.

CharacterSet

An optional integer value which specifies the character set for the message text. If this parameter is

omitted, the default is for the message to be composed using the standard UTF-8 character set.

EncodingType

An optional integer value which specifies the content encoding to use for the message text. The default is for the control to use 8-bit encoding. One of the following values may be used:

Value	Description
mimeEncoding7Bit	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the first bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
mimeEncoding8Bit	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding is used with UTF-8 and other multi-byte character sets,
mimeEncodingQuoted	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
mimeEncodingBase64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

email addresses may be specified as simple addresses, or as commented addresses that include the sender's name or other information. For example, any one of these address formats are acceptable:

```
user@domain.tld
User Name <user@domain.tld>
user@domain.tld (User Name)
```

To specify multiple addresses, you should separate each address by a comma or semi-colon. Note that the ***From*** parameter cannot specify multiple addresses, however it is permitted with the ***To***, ***Cc*** and ***Bcc*** parameters.

To send a message that contains HTML, it is recommended that you provide both a plain text version of the message body and an HTML formatted version. While it is permitted to send a message that only contains HTML, some older mail clients may not be capable of displaying the message correctly. In some cases, anti-spam software will increase the spam score of messages that do not contain a plain text message body. This can result in your message being rejected or quarantined by the mail

server.

Example

```
nError = MailMessage1.ComposeMessage(editFrom.Text, _
                                     editTo.Text, _
                                     editCc.Text, _
                                     editBcc.Text, _
                                     editSubject.Text, _
                                     editMessage.Text)
If nError > 0 Then
    MessageBox MailMessage1.LastErrorString, vbExclamation
    Exit Sub
End If
```

See Also

Bcc Property, Cc Property, Encoding Property, From Property, Recipient Property, Recipients Property, Subject Property, Text Property, To Property

CreatePart Method

Create a new message part in a multipart message.

Syntax

object.**CreatePart**([*MessageText*], [*CharacterSet*], [*EncodingType*])

Parameters

MessageText

An optional string argument which specifies the body of the new message part. Each line of text contained in the string should be terminated with a carriage-return/linefeed (CRLF) pair, which is recognized as the end-of-line. If the argument is not specified, then the message part will have an empty body.

CharacterSet

An optional integer value which specifies the character set for the message part. If this parameter is omitted, the default is for the message to be composed using the standard UTF-8 character set.

EncodingType

An optional integer value which specifies the content encoding to use for the message part. The default is for the control to use 7-bit encoding. If an 8-bit character set is specified for the ***CharacterSet*** argument, the default encoding type will be set to quoted-printable. One of the following values may be used:

Value	Description
mimeEncoding7Bit	Each character is encoded in one or more bytes, with each byte being 8 bits long, with the first bit cleared. This encoding is most commonly used with plain text using the US-ASCII character set, where each character is represented by a single byte in the range of 20h to 7Eh.
mimeEncoding8Bit	Each character is encoded in one or more bytes, with each byte being 8 bits long and all bits are used. 8-bit encoding may be used with UTF-8 and other multi-byte character sets.
mimeEncodingQuoted	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
mimeEncodingBase64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CreatePart** method creates a new message part. If the current message is a simple RFC822 formatted message, then this method converts it to a MIME multipart message. The current message part will be set to the new part that was just created.

See Also

Bcc Property, Cc Property, Encoding Property, From Property, Subject Property, Text Property, To Property

DecodeText Method

Decode a string which was previously encoded using base64 or quoted-printable encoding.

Syntax

object.**DecodeText**(*EncodedText*, *MessageText*, [*CharacterSet*], [*EncodingType*])

Parameters

EncodedText

A string which contains the encoded text which should be decoded.

MessageText

A string variable passed by reference which will contain the decoded text when the method returns.

CharacterSet

An optional integer value which specifies the character set to use when decoding the encoded text. If this value does not match the character set used when the text was originally encoded, the resulting output text may be invalid. If no character set is specified, this method will default to using UTF-8.

EncodingType

An optional integer value which specifies the content encoding to use when decoding the text. It may be one of the following values:

Value	Description
mimeEncodingQuoted	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
mimeEncodingBase64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text. This is the default encoding type used by this method.

Return Value

The method returns the number of characters of decoded text. A return value of zero indicates no text has been decoded. If the method fails, it will return -1 and the **LastError** property can be used to determine the cause of the failure. In most cases where the method fails, it is because an invalid character set or encoding type has been specified.

Remarks

This method provides a means to decode text that was previously encoded using either base64 or quoted-printable encoding. In most cases, it is not necessary to use this method because the message

parser will detect which character set and encoding was used, then automatically decode the message text if necessary.

The value of the ***CharacterSet*** parameter does not affect the resulting output text, it is only used when decoding the input text. The previous contents of the ***MessageText*** string will be replaced by the decoded text, and the output string will always be Unicode.

If the ***CharacterSet*** parameter is specified as **mimeCharsetUTF16**, the encoding type must be **mimeEncodingBase64**. Other encoding methods are not supported for Unicode strings and will cause the method to fail. In most cases, it is preferable to always use **mimeEncodingBase64** as the encoding method, with quoted-printable encoding only used for legacy support. If an unsupported encoding type is specified, this method will return -1 and the output text string will be empty. This method cannot be used to decode uuencoded text.

If you are developing your application using Visual Basic 6.0, the IDE does not provide complete support for Unicode. The decoded text may appear to be corrupted when examining it in the debugger. This is because the IDE will attempt to convert the string to ANSI using the system default code page. To display Unicode text correctly, you must use controls which are Unicode aware, such as the Microsoft InkEdit control.

See Also

Encoding Property, Text Property, ComposeMessage Method, CreatePart Method, EncodeText Method

DeleteHeader Method

Delete a header field from the current message part.

Syntax

object.**DeleteHeader**(*HeaderField*)

Parameters

HeaderField

A string which specifies the header field to delete from the current message part.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **DeleteHeader** method deletes the specified header field value from the current message part.

See Also

HeaderField Property, HeaderValue Property, Part Property, PartCount Property, GetHeader Method, SetHeader Method

DeleteMessage Method

Remove the specified message from the current message store.

Syntax

object.**DeleteMessage**([*MessageIndex*])

Parameters

MessageIndex

An integer value which specifies the message that is to be removed from the message store. If this parameter is omitted, the current message as specified by the value of the **StoreIndex** property will be used.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **DeleteMessage** method marks the specified message for deletion from the storage file. When the message store is closed or purged, the message is removed from the file. Once a message has been marked for deletion, it may no longer be referenced by the application. For example, you cannot access the contents of a message that has been deleted.

The message store must be opened with write access. This method will fail if you attempt to delete a message from a storage file that has been opened for read-only access. If the application needs to delete messages in the message store, it is recommended that the file be opened for exclusive access using the **mimeStoreLock** option when calling the **OpenStore** method.

See Also

StoreCount Property, StoreIndex Property, PurgeStore Method, ReadStore Method, ReplaceMessage Method, WriteStore Method

DeletePart Method

Delete the specified message part in the current message.

Syntax

object.**DeletePart**([*MessagePart*])

Parameters

MessagePart

An optional integer value which specifies the message part to remove from a multipart message.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **DeletePart** method deletes the specified message part in the current message. If the optional message part is not specified, then the current message part is deleted. This method cannot be used to delete the main body of the message. Use the **ClearMessage** method to clear the contents of the complete message.

See Also

AttachFile Method, ClearMessage Method, CreatePart Method

EncodeText Method

Encodes a string using base64 or quoted-printable encoding.

Syntax

object.**EncodeText**(*MessageText*, *EncodedText*, [*CharacterSet*], [*EncodingType*])

Parameters

MessageText

A string that contains the text which will be encoded.

EncodedText

A string variable passed by reference which will contain the encoded text when the method returns.

CharacterSet

An optional integer value which specifies the character set to use when encoding the text. If no character set is specified, this method will default to using UTF-8.

EncodingType

An optional integer value which specifies the content encoding type. It may be one of the following values:

Value	Description
mimeEncodingQuoted	Quoted-printable encoding is designed for textual messages where most of the characters are represented by the ASCII character set and is generally human-readable. Non-printable characters or 8-bit characters with the high bit set are encoded as hexadecimal values and represented as 7-bit text. Quoted-printable encoding is typically used for messages which use character sets such as ISO-8859-1, as well as those which use HTML.
mimeEncodingBase64	Base64 encoding converts binary or text data to ASCII by translating it so each base64 digit represents 6 bits of data. This encoding method is commonly used with messages that contain binary data (such as binary file attachments), or when text uses extended characters that cannot be represented by 7-bit ASCII. It is recommended that you use base64 encoding with Unicode text. This is the default encoding type used by this method.

Return Value

The method returns the number of characters of encoded text. A return value of zero indicates no text has been encoded. If the method fails, it will return -1 and the **LastError** property can be used to determine the cause of the failure. In most cases where the method fails, it is because an invalid character set or encoding type has been specified.

Remarks

This method provides a means to encode text using either base64 or quoted-printable encoding. It is not necessary to use this method to encode text when assigning a value to the **Text** property. The control will automatically encode message text which contains non-ASCII characters using the character set specified when the message is created.

If the *CharacterSet* parameter s specified, the method will convert the message text using the ANSI code page associated with the character set, and then the text will be encoded. If the parameter is omitted, the message text will be converted to UTF-8 and then encoded.

If the **mimeCharsetUTF16** character set is specified, you must also specify **mimeEncodingBase64** as the encoding method. Other encoding methods are not supported and this will cause the method to fail. It is not recommended you encode text as UTF-16 unless there is a specific requirement to use that character set.

It is recommended that you use the **mimeCharsetUTF8** character set whenever possible. It is capable of encoding all Unicode code points, and is a standard for virtually all modern Internet applications. In most cases, it is preferable to use **mimeEncodingBase64** as the encoding method, with quoted-printable encoding only used for legacy support.

If you are developing your application using Visual Basic 6.0, the IDE does not provide complete support for Unicode. The decoded text may appear to be corrupted when examining it in the debugger. This is because the IDE will attempt to convert the string to ANSI using the system default code page. To display Unicode text correctly, you must use controls which are Unicode aware, such as the Microsoft InkEdit control.

See Also

Encoding Property, Text Property, ComposeMessage Method, CreatePart Method, DecodeText Method

ExportMessage Method

Export the current message to a file on the local system.

Syntax

object.**ExportMessage**(*FileName*, [*Options*])

Parameters

FileName

A string which specifies the name of the file that will contain the message. If the file does not exist, it will be created. If it does exist, it will be overwritten with the contents of the message.

Options

An optional integer value which specifies one or more options. If this argument is omitted, the **Options** property value will be used as the default. The following values may be combined using a bitwise Or operator:

Value	Description
mimeOptionDefault	The default export options. The headers for the message are written out in a specific consistent order, with custom headers written to the end of the header block regardless of the order in which they were set or imported from another message. If the message contains Bcc, Received, Return-Path, Status or X400-Received header fields, they will not be exported.
mimeOptionAllHeaders	All headers, including the Bcc, Received, Return-Path, Status and X400-Received header fields will be exported. Normally these headers are not exported because they are only used by the mail transport system. This option can be useful when exporting a message to be stored on the local system, but should not be used when exporting a message to be delivered to another user.
mimeOptionKeepOrder	The original order in which the message header fields were set or imported are preserved when the message is exported.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

See Also

ImportMessage Method

ExtractAllFiles Method

Extract all file attachments from the current message, storing them in the specified directory.

Syntax

object.**ExtractAllFiles**([*Directory*])

Parameters

Directory

An optional string that specifies the name of the directory where the file attachments should be stored. If this parameter is omitted or points to an empty string, the attached files will be stored in the current working directory on the local system.

Return Value

If the method succeeds, the return value is the number of file attachments which were extracted from the current message. If the message does not contain any file attachments, this method will return a value of zero. If the method fails, the return value is -1. To get extended error information, check the value of the **LastError** property.

Remarks

This method will extract all of the files that are attached to the current message and store them in the specified directory. The directory must exist and the current user must have the appropriate permissions to create files there. If a file with the same name as the attachment already exists, it will be overwritten with the contents of the attachment. If the file attachment was encoded using base64 or uuencode, this method will automatically decode the contents of the attachment.

To store a file attachment on the local system using a name that is different than the file name of the attachment, use the **ExtractFile** method.

See Also

Attachment Property, AttachData Method, AttachFile Method, ExtractFile Method

ExtractFile Method

Extract the contents of a file attachment and store it on the local system.

Syntax

object.**ExtractFile**(*FileName*, [*MessagePart*])

Parameters

FileName

A string which specifies the name of the file that the attachment will be written to. If the file does not exist, it will be created. If the file exists, it will be overwritten.

MessagePart

An optional integer value that specifies the message part number that contains the file attachment. If this parameter is omitted, the method will extract the file attachment in the current message part.

Return Value

A value of zero is returned if the action was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ExtractFile** method writes the contents of a message part, typically a file attachment, to a file on the local system. This method will automatically decode any binary file attachments. To determine if the current message part contains a file attachment, check the value of the **Attachment** property.

See Also

Attachment Property, AttachData Method, AttachFile Method, ExtractAllFiles Method, FindAttachment Method

FindAttachment Method

Search the current message for a file attachment with the specified file name.

Syntax

object.**FindAttachment**(*FileName*)

Parameters

FileName

A string the specifies the name of the file attachment to search for. This parameter should only specify a base file name; it should not include a file path and cannot be an empty string.

Return Value

If the method succeeds, the return value is the message part number that contains the file attachment. If the message does not contain an attachment that matches the specified file name, the return value is -1. To get extended error information, check the value of the **LastError** property.

Remarks

The **FindAttachment** method will search the current message for a attachment that matches the specified file name. The search is not case-sensitive, however it must match the attachment file name completely. This method will not match partial file names or names that include wildcard characters.

Example

```
nMessagePart = MailMessage1.FindAttachment(strFileName)

If nMessagePart > -1 Then
    MailMessage1.ExtractFile(strFileName, nMessagePart)
End If
```

See Also

Attachment Property, AttachFile Method, ExtractAllFiles Method, ExtractFile Method

FindMessage Method

Search for a message in the current message store.

Syntax

object.**FindMessage**(*HeaderField*, *HeaderValue*, [*MessageIndex*], [*Options*])

Parameters

HeaderField

A string which specifies the name of the header field that should be searched. The header field name is not case sensitive.

HeaderValue

A string which specifies the header value that should be searched for. The search options can be used to specify if the search is case-sensitive, and whether the search should return partial matches to the string.

MessageIndex

An optional integer value which specifies the message number that should be used when starting the search. If this parameter is omitted, the search will begin with the first message in the message store.

Options

An optional integer value which specifies how the search will be performed. If this parameter is omitted, the default search options will be used. One or more of the following values may be specified:

Value	Description
mimeSearchDefault	Perform a complete match against the specified header value. The comparison is not case-sensitive. It is the default search option used if this parameter is omitted.
mimeSearchCaseSensitive	The header value comparison will be case-sensitive. Note that this does not affect header field names. Matches for header names are always case-insensitive.
mimeSearchPartialMatch	Perform a partial match against the specified header value. It recommended that this option be used when searching for matches to email addresses.
mimeSearchDecodeHeaders	Decode any encoded message headers before comparing them to the specified value. This option can increase the amount of time required to search the message store and should only be used when necessary.

Return Value

If the method is successful, it returns the message number which specifies the message that matches the search criteria. If no matching message could be found, the method will return zero.

Remarks

The **FindMessage** method is used to search the message store for a message which matches a specific header field value. For example, it can be used to find every message which is addressed to a specific recipient or has a subject which matches a particular string value.

See Also

StoreCount Property, StoreIndex Property, CloseStore Method, OpenStore Method, ReadStore Method, WriteStore Method

GetFirstHeader Method

Return the first header in the current message part.

Syntax

object.**GetFirstHeader**(*HeaderField*, *HeaderValue*)

Parameters

HeaderField

A string which will contain the name of the first header field when the method returns. This parameter must be passed by reference.

HeaderValue

A string which will contain the value of the first header field when the method returns. This parameter must be passed by reference.

Return Value

A value of True is returned if the first header value was returned. If the current message part does not contain any header fields, this method will return False.

Example

The following example enumerates all of the headers in the main part of the current message and adds them to a listbox:

```
Dim strHeader As String, strValue As String
Dim bResult As Boolean

bResult = MailMessage1.GetFirstHeader(strHeader, strValue)
Do While bResult
    List1.AddItem strHeader & ": " & strValue
    bResult = MailMessage1.GetNextHeader(strHeader, strValue)
Loop
```

See Also

Part Property, GetHeader Method, GetNextHeader Method, SetHeader Method

GetHeader Method

Return the value for the specified header in the current message part.

Syntax

object.**GetHeader**(*HeaderField*, *HeaderValue*)

Parameters

HeaderField

A string variable which will specifies the name of the header field to return the value of. Header field names are not case sensitive.

HeaderValue

A string variable which will contain the value of the specified header field.

Return Value

A value of True is returned if the header value was returned. If the current message part does not contain the specified header field, this method will return False.

Parameters

The **GetHeader** method is used to retrieve the value for a specific header in the current message part. If there are multiple headers with the same name, the first value will be returned. To enumerate all of the headers in a message, including duplicate header fields, use the **GetFirstHeader** and **GetNextHeader** methods.

See Also

AllHeaders Property, Part Property, GetFirstHeader Method, GetNextHeader Method, SetHeader Method

GetNextHeader Method

Return the next header in the current message part.

Syntax

object.**GetNextHeader**(*HeaderField*, *HeaderValue*)

Parameters

HeaderField

A string which will contain the name of the next header field when the method returns. This parameter must be passed by reference.

HeaderValue

A string which will contain the value of the next header field when the method returns. This parameter must be passed by reference.

Return Value

A value of True is returned if the next header value was returned. If there are no more header fields in the current message part, this method will return False.

Example

The following example enumerates all of the headers in the main part of the current message and adds them to a listbox:

```
Dim strHeader As String, strValue As String
Dim bResult As Boolean

bResult = MailMessage1.GetFirstHeader(strHeader, strValue)
Do While bResult
    List1.AddItem strHeader & ": " & strValue
    bResult = MailMessage1.GetNextHeader(strHeader, strValue)
Loop
```

See Also

Part Property, GetFirstHeader Method, GetHeader Method, SetHeader Method

ImportMessage Method

Replace the current message with the contents of a file.

Syntax

object.**ImportMessage**(*FileName*)

Parameters

FileName

A string which specifies the name of the text file to import.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

See Also

AppendMessage Method, ExportMessage Method, ParseMessage 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 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 objMessage = CreateObject("SocketTools.MailMessage.11")

nError = objMessage.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

Uninitialize Method

OpenStore Method

Open the specified message storage file.

Syntax

object.**OpenStore**(*FileName*, [*OpenMode*])

Parameters

FileName

A string which specifies the name of the storage file.

OpenMode

An optional integer value which specifies how the storage file will be opened. If this parameter is omitted, the file will be opened for read-only access. One or more of the following values may be specified:

Value	Description
mimeStoreRead	The message store will be opened for read access. The contents of the message store can be accessed, but cannot be modified by the process unless it has also been opened for writing.
mimeStoreWrite	The message store will be opened for writing. This mode also implies read access and must be specified if the application needs to modify the contents of the message store.
mimeStoreCreate	The message store will be created if the storage file does not exist. If the file exists, it will be truncated. This mode implies read and write access.
mimeStoreLock	The message store will be opened so that it may only be accessed and modified by the current process.
&H1000	mimeStoreCompress	The contents of the message store are compressed. This option is automatically enabled if a compressed message store is opened for reading or writing.
&H2000	mimeStoreMailbox	The message store should use the UNIX mbox format when reading and storing messages. This option is provided for backwards compatibility and is not recommended for general use.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **OpenStore** method opens a message storage file which contains one or more messages. If the storage file is opened for read access, the application can search the message store and extract messages but it cannot add or delete messages. To add new messages or delete existing messages from the store, it must be opened with write access.

The message store is designed to be a simple, effective way to store multiple messages together in a single file. When the message store is opened, the contents are indexed in memory. Although there is no specific limit to the number of messages that can be stored, there must be sufficient memory available to build an index of each message and its headers. If the application must store and manage a very large number of messages, it is recommended that you use a database rather than a flat-file message store.

Message Store Format

Each message is prefixed by a control sequence of five ASCII 01 characters followed by an ASCII 10 and ASCII 13 character. The messages themselves are stored unmodified in their original text format. The length of each message is calculated based on the location of the control sequence that delimits each message, and explicit message lengths are not stored in the file. This means that it is safe to manually change the message contents, as long as the message delimiters are preserved.

If the message store is compressed, the contents of the storage file are expanded when the file is opened and then re-compressed when the storage file is closed. Using the **mimeStoreCompress** option reduces the size of the storage file and prevents the contents of the message store from being read using a text file editor. However, enabling compression will increase the amount of memory allocated by the control and can increase the amount of time that it takes to open and close the storage file.

The control also has a backwards compatibility mode where it will recognize storage files that use the UNIX mbox format. While this format is supported for accessing existing files, it is not recommended that you use it when creating new message stores or adding messages to an existing store. There are a number of different variants on the mbox format that have been used by different Mail Transfer Agents (MTAs) on the UNIX platform. For example, the mboxrd variant looks identical to the mboxcl2 variant, and they are programmatically indistinguishable from one another, but they are not compatible. For this reason, the use of the mbox format is strongly discouraged.

See Also

StoreCount Property, StoreFile Property, CloseStore Method, FindMessage Method, ReadStore Method, WriteStore Method

ParseAddress Method

Parse an Internet email address.

Syntax

object.**ParseAddress**(*Address*)

Parameters

Address

A string which specifies the email address to be parsed.

Return Value

A string containing the parsed address is returned if method was successful, otherwise an empty string is returned.

Remarks

The **ParseAddress** method parses a string which contains an email address and returns only the address portion, excluding any comments. An address may contain comments enclosed in parenthesis, or may specify a name along with the address in which case the address is enclosed in angle brackets. For example, consider the following header field value:

```
"User Name" <user@domain.com> (This is a comment)
```

The **ParseAddress** method would return "user@domain.com" if passed the above string, removing the name and any comments. Note that the **ParseAddress** method will only parse a single address. If multiple addresses are specified, they must be comma delimited and split prior to calling this method.

Example

The following example parses all of the recipient email addresses in the current message, storing them in the *strAddresses* string array.

```
Dim strAddresses() As String, strAddress As String
Dim nIndex As Integer, nAddresses As Integer

nAddresses = 0
strAddresses = Split(MailMessage1.To & "," & MailMessage1.Cc, ",")

For nIndex = 0 To UBound(strAddresses)
    If Len(Trim(strAddresses(nIndex))) > 0 Then
        strAddress = MailMessage1.ParseAddress(strAddresses(nIndex))
        If Len(strAddress) > 0 Then
            strAddresses(nAddresses) = strAddress
            nAddresses = nAddresses + 1
        End If
    End If
Next
```

See Also

Cc Property, From Property, To Property

ParseMessage Method

Parse the specified string, adding the contents to the current message.

Syntax

object.**ParseMessage**(*MessageText*)

Parameters

MessageText

A string that contains the message text to be parsed.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ParseMessage** method parses a string which contains message data, adding it to the current message. This method is useful when the application needs to parse an arbitrary block of text and add it to the current message. If the string contains header fields, the values will be added to the message header. Once the end of the header block is detected, all subsequent text is added to the body of the message.

Note that unlike the **ImportMessage** method, the **ParseMessage** method does not clear the contents of the current message and may be called multiple times. Use the **ClearMessage** method to clear the current message before calling **ParseMessage** if necessary.

Example

The following example demonstrates the use of **ParseMessage** to parse multiple blocks of data from a file. This example effectively does the same thing as calling the **ImportMessage** method:

```
MailMessage1.ClearMessage

hFile = FreeFile()
Open strFileName For Input As hFile
nFileLength = LOF(hFile)

Do While nFileLength > 0
    '
    ' Read the contents of the file in 1K blocks; note that
    ' this is intentionally inefficient to demonstrate
    ' multiple calls to the ParseMessage method.
    '
    cbBuffer = nFileLength: If cbBuffer > 1024 Then cbBuffer = 1024
    nFileLength = nFileLength - cbBuffer
    strBuffer = Input(cbBuffer, hFile)
    '
    ' Parse the string, adding to the current message
    '
    nError = MailMessage1.ParseMessage(strBuffer)
    If nError > 0 Then
        MsgBox MailMessage1.LastErrorString, vbExclamation
        Exit Do
    End If
Loop
```

```
Close hFile
```

See Also

ClearMessage Method, ImportMessage Method

PurgeStore Method

Purge all deleted messages from the current message store.

Syntax

object.**PurgeStore**

Parameters

None.

Return Value

A value of zero is returned if the action was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **PurgeStore** method purges all deleted messages from the message store. If the storage file has been opened in read-only mode or there are no messages marked for deletion, this method will take no action.

When the **CloseStore** method is called, the storage file will automatically be purged. To prevent deleted messages from being removed from the message store, use the **Reset** method.

See Also

StoreFile Property, CloseStore Method, DeleteMessage Method, OpenStore Method, ReadStore Method, WriteStore Method

ReadStore Method

Retrieve a message from the message store, replacing the current message.

Syntax

object.**ReadStore**(*MessageIndex*)

Parameters

MessageIndex

An integer value which specifies the message that is to be removed from the message store. If this parameter is omitted, the current message as specified by the value of the **StoreIndex** property will be used.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ReadStore** method reads the specified message from the message store, and the contents of that message will replace the current message. This method will update the current message index in the storage file.

See Also

StoreCount Property, StoreFile Property, StoreIndex Property, CloseStore Method, DeleteMessage Method, FindMessage Method, ReplaceMessage Method, WriteStore Method

ReplaceMessage Method

Replace the specified message in the current message store.

Syntax

object.**ReplaceMessage**([*MessageIndex*])

Parameters

MessageIndex

An integer value which specifies the message that is to be replaced in the message store. If this parameter is omitted, the current message as specified by the value of the **StoreIndex** property will be used.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **ReplaceMessage** method replaces the specified message with a new message. The message number may be a message that has been previously marked for deletion. It is important to note that the change will not be reflected in the physical storage file until it has been closed. This method will update the current message index in the storage file.

The message store must be opened with write access. This method will fail if you attempt to replace a message from a storage file that has been opened for read-only access. If the application needs to replace messages in the message store, it is recommended that the file be opened for exclusive access using the **mimeStoreLock** option when calling the **OpenStore** method.

See Also

StoreCount Property, StoreIndex Property, DeleteMessage Method, PurgeStore Method, ReadStore Method, WriteStore 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 and any handles allocated by the control will be released. If a message store has been opened, it will be closed. If messages had been flagged for deletion from the current message store, they will not be purged.

See Also

Initialize Method, Uninitialize Method

SetHeader Method

Set the value for the specified header in the current message part.

Syntax

object.**SetHeader**(*HeaderField*, *HeaderValue*)

Parameters

HeaderField

A string which specifies the name of the header field to create or modify. If the header field does not exist, then it will be created. If the header field exists, the current value will be overwritten.

HeaderValue

A string which specifies the value of the specified header field.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

See Also

AllHeaders Property, Part Property, GetFirstHeader Method, GetHeader Method, GetNextHeader Method

Uninitialize Method

Uninitialize the control and reset it to its default state.

Syntax

object.**Uninitialize**

Parameters

None.

Return Value

None.

Remarks

The **Uninitialize** method frees the memory allocated for the current message 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

WriteStore Method

Store the current message in the message store.

Syntax

object.**WriteStore**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **WriteStore** method will always append the current message to the storage file. If you want to replace a message in the message store, you should use the **ReplaceMessage** method. This method will update the value of the **StoreIndex** property to specify the message number for the new message that has been added to the storage file.

See Also

StoreCount Property, StoreFile Property, StoreIndex Property, CloseStore Method, DeleteMessage Method, FindMessage Method, ReadStore Method, ReplaceMessage Method

Mail Message Control Events

Event	Description
OnError	This event is generated when a control error occurs

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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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

Message Character Sets

Value	Name	Code Page	Description
mimeCharsetUSASCII	us-ascii	A character set which defines 7-bit printable characters with values ranging from 20h to 7Eh. An application that uses this character set has the broadest compatibility with most mail servers (MTAs) because it does not require the server to handle 8-bit characters correctly when the message is delivered.	
mimeCharsetISO8859_1	iso-8859-1	A character set for most western European languages such as English, French, Spanish and German. This character set is also commonly referred to as Latin-1. This character set is similar to Windows code page 1252 (Windows-1252), however there are differences such as the Euro symbol.	
mimeCharsetISO8859_2	iso-8859-2	A character set for most central and eastern European languages such as Czech, Hungarian, Polish and Romanian. This character set is also commonly referred to as Latin-2. This character set is similar to Windows code page 1250, however the characters are arranged differently.	
mimeCharsetISO8859_3	iso-8859-3	A character set for southern European languages such as Maltese and Esperanto. This character set was also used with the Turkish language, but it was superseded by ISO 8859-9 which is the preferred character set for Turkish. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead.	
mimeCharsetISO8859_4	iso-8859-4	A character set for northern European languages such as Latvian, Lithuanian and Greenlandic. This character set is	

		not widely used in mail messages and it is recommended that you use UTF-8 instead.
mimeCharsetISO8859_5	iso-8859-5	A character set for Cyrillic languages such as Russian, Bulgarian and Serbian. This character set was never widely adopted and most mail messages use either KOI8 or UTF-8 encoding.
mimeCharsetISO8859_6	iso-8859-6	A character set for Arabic languages. Note that the application is responsible for displaying text that uses this character set. In particular, any display engine needs to be able to handle the reverse writing direction and analyze the context of the message to correctly combine the glyphs.
mimeCharsetISO8859_7	iso-8859-7	A character set for the Greek language. This character set is also commonly referred to as Latin/Greek. This character set is no longer widely used and has largely been replaced with UTF-8 which provides more complete coverage of the Greek alphabet.
mimeCharsetISO8859_8	iso-8859-8	A character set for the Hebrew language. Note that similar to Arabic, Hebrew uses a reverse writing direction. An application which displays this character should be capable of processing bi-directional text where a single message may include both right-to-left and left-to-right languages, such as Hebrew and English. In most cases it is recommended that you use UTF-8 instead of this character set.
mimeCharsetISO8859_9	iso-8859-9	A character set for the Turkish language. This character set is also commonly referred to as Latin-5. This character set is nearly identical to ISO 8859-1, except that it replaces certain Icelandic characters with Turkish characters.

mimeCharsetISO8859_10	iso-8859-10	A character set for the Danish, Icelandic, Norwegian and Swedish languages. This character set is also commonly referred to as Latin-6 and is similar to ISO 8859-4.
mimeCharsetISO8859_13	iso-8859-13	A character set for Baltic languages. This character set is also commonly referred to as Latin-7. This character set is similar to ISO 8859-4, except it adds certain Polish characters and does not support Nordic languages.
mimeCharsetISO8859_14	iso-8859-14	A character set for Gaelic languages such as Irish, Manx and Scottish Gaelic. This character set is also commonly referred to as Latin-8. This character set replaced ISO 8859-12 which was never fully implemented.
mimeCharsetISO8859_15	iso-8859-15	A character set for western European languages. This character set is also commonly referred to as Latin-9 and is nearly identical to ISO8859-1 except that it replaces lesser-used symbols with the Euro sign and some letters.
mimeCharsetISO2022_JP	iso-2022-jp	A multi-byte character encoding for Japanese that is widely used with mail messages. This is a 7-bit encoding where all characters start with ASCII and uses escape sequences to switch to the double-byte character sets.
mimeCharsetISO2022_KR	iso-2022-kr	A multi-byte character encoding for Korean which encodes both ASCII and Korean double-byte characters. This is a 7-bit encoding which uses the shift in and shift out control characters to switch to the double-byte character set.
mimeCharsetISO2022_CN	x-cp50227	A multi-byte character encoding for Simplified Chinese which encodes both ASCII and Chinese double-byte characters. This is a 7-bit encoding which uses the shift in

		and shift out control characters to switch to the double-byte character set.
mimeCharsetKOI8R	koi8-r	A character set for Russian using the Cyrillic alphabet. This character set also covers the Bulgarian language. Most mail messages in the Russian language use this character set or UTF-8 instead of ISO 8859-5, which was never widely adopted.
mimeCharsetKOI8U	koi8-u	A character set for Ukrainian using the Cyrillic alphabet. This character set is similar to the KOI8-R character set, but replaces certain symbols with Ukrainian letters. Most mail messages in the Ukrainian language use this character set or UTF-8 instead of ISO 8859-5, which was never widely adopted.
mimeCharsetGB2312	x-cp20936	A multi-byte character encoding which can represent ASCII and simplified Chinese characters. It has been superseded by GB18030, however it remains widely used in China.
mimeCharsetGB18030	gb18030	A Unicode transformation format which can represent all Unicode code points and supports both simplified and traditional Chinese characters. It is backwards compatible with GB2312 and supersedes that character set.
mimeCharsetBIG5	big5	A multi-byte character set that supports both ASCII characters and traditional Chinese characters. It is widely used in Taiwan, Hong Kong and Macau. It is no longer commonly used in China, which has developed GB18030 as a standard encoding. Microsoft's implementation of Big5 on Windows does not support all of the extensions and is missing certain code points.
mimeCharsetUTF7	utf-7	A Unicode transformation format

		that uses variable-length character encoding to represent Unicode text as a stream of ASCII characters that are safe to transport between mail servers that only support 7-bit printable characters. It is primarily used as an alternative to UTF-8 when quoted-printable or base64 encoding is not desired.	
mimeCharsetUTF8	utf-8	A Unicode transformation format that uses multi-byte character sequences to represent Unicode text. It is backwards compatible with the ASCII character set, however because it uses 8-bit text, it is recommended that you use either quoted-printable or base64 encoding to ensure compatibility with mail servers that do not support 8-bit characters.	
mimeCharsetUTF16	utf-16le	N/A	A 16-bit Unicode format that represents each character as a 16-bit value in little endian byte order. This character set is not widely used in mail messages and it is recommended that you use UTF-8 instead. UTF-16 characters in big endian byte order are not supported.

Remarks

When composing a new message, it is recommended that you always use UTF-8 as the character set encoding which ensures broad compatibility with most applications. The other character sets are primarily used when parsing messages generated by other applications. Internally, all message headers and text are processed as UTF-8 and returned as Unicode strings.

In addition to the character sets listed above, the control will recognize additional character sets which correspond to specific Windows code pages, as well several variants. These additional character sets are included for compatibility with other applications; they are not defined because they should not be used when composing new messages.

It is important to note that while certain Windows character sets are similar to standard ISO character sets, they are not identical. For example, although the Windows-1252 character set is nearly identical to ISO 8859-1, they are not interchangeable. Some legacy applications make the error of representing Windows ANSI character sets as 8-bit ISO character sets, which can result in errors when converting them to Unicode. This is something to be aware of when encoding and decoding text generated by

older applications. Before the widespread adoption of UTF-8, it was particularly common for legacy Windows mail clients to default to using Windows-1252 for text and label it as using ISO 8859-1.

Although the control supports UTF-16, it is recommended you use UTF-8 instead. Text which uses UTF-16 will always be base64 encoded, and some mail clients may not recognize it as a valid character set. If the message does not specify if big endian or little endian byte order is used, the library will default to little endian. When UTF-16 is used when composing a new message, it will always use little endian byte order.

If you are using this control with Visual Basic 6.0, be aware that the IDE does not provide complete support for Unicode text. Although the control uses Unicode internally, if a header or message body contains characters which cannot be displayed using the current system ANSI code page, the text can appear to be corrupted when examining the string using the debugger. If a message contains text which uses a character set other than the system default, you must use controls which are Unicode aware to display the text, such as the Microsoft InkEdit control. The standard TextBox and other common controls in Visual Basic do not support Unicode.

See Also

ComposeMessage Method, CreatePart Method, DecodeText Method, EncodeText Method

News Feed Control

Retrieve and process the contents of a syndicated news feed.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	NewsFeedCtl.NewsFeed
File Name	CSRSSX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.NewsFeed.11
ClassID	D82CEE60-9C78-4F37-BD5A-E8A34B438AD9
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None

Overview

Really Simple Syndication (RSS) is a collection of standardized formats that are used to publish information about content that is frequently changed. A news feed is published in XML format, which contains one or more items that includes summary text, hyperlinks to source content and additional metadata that is used to describe the item. News feeds can be used for a variety of purposes, including providing updates for weblogs, news headlines, video and audio content. RSS can also be used for other purposes, such as a software updates, where new updates are listed as items in the feed.

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

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop

and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

News Feed Control Properties

Property	Description
Category	Gets the category or categories that the news feed channel belongs to
Copyright	Gets the copyright notice for the news feed
Description	Gets a description of the news feed channel
Editor	Gets the email address of the person responsible for managing the content of the feed
FeedVersion	Gets the version of the news feed
Generator	Gets the name of the application that generated the news feed
ImageLink	Gets the hyperlink for the image associated with the news feed
ImageTitle	Gets the title for the image associated with the news feed
ImageUrl	Gets the URL for the image associated with the news feed
IsInitialized	Determine if the control has been initialized
ItemAuthor	Gets the email address of the person who authored the current news item
ItemComments	Gets the URL that links to further discussion about the current news item
ItemCount	Gets the number of news items in the feed channel
ItemEnclosure	Gets the URL that links to a file associated with the current news item
ItemGuid	Gets a value that uniquely identifies the current news item in the feed channel
ItemId	Gets the numeric ID for the current news item
ItemLink	Gets the URL that links to additional information about the current news item
ItemPublished	Gets the date and time that the current news item was published
ItemSource	Gets a URL that links to the original news feed that contained the current item
ItemText	Gets the text that describes the current news item
ItemTitle	Gets the title of the current news item
Language	Gets the language that the news feed channel is written in, using standard language codes
LastBuild	Gets the date and time that the news feed was last modified
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error to occur
LinkUrl	Gets the URL that links to the website corresponding to the news feed
LocalFeed	Return whether the news feed was loaded from a file on the local system or a server
Published	Gets the date and time that the news feed was published
ThrowError	Enable or disable error handling by the container of the control
Timeout	Gets and sets the amount of time until a blocking operation fails
TimeToLive	Gets a value which specifies the frequency in seconds at which the feed should be refreshed
Title	Gets the title of the news feed

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
URL	Gets the URL for the current news feed
Version	Gets the current version of the object
Webmaster	Gets the email address of the person responsible for technical issues related to the news feed

Category Property

Gets a value which specifies the category or categories that the channel belongs to.

Syntax

object.**Category**

Remarks

The **Category** property is used to identify the category that the news feed channel belongs to. This is an optional value that may not be defined, in which case the property will return an empty string.

Data Type

String

See Also

Description Property, Language Property, LinkUrl Property, Published Property, Title Property

Copyright Property

Gets a value which specifies a copyright notice for the content.

Syntax

object.**Copyright**

Remarks

The **Copyright** property returns a string which contains a copyright notice for the news feed. If the feed does not specify a copyright, then this property will return an empty string.

Data Type

String

See Also

Description Property, Language Property, LinkUrl Property, Published Property, Title Property

Description Property

Gets a value which describes the news feed channel.

Syntax

object.**Description**

Remarks

The **Description** property returns a string that provides an overview of the news feed and the type of information that is provided. If a description of the feed has not been specified, this property will return an empty string. Note that a news feed which conforms to the standard specification requires a description.

Data Type

String

See Also

Language Property, LinkUrl Property, Published Property, Title Property

Editor Property

Gets a value which identifies the person responsible for managing the content of the news feed.

Syntax

object.**Editor**

Remarks

The **Editor** property returns a string that identifies the person responsible for managing the content of the news feed. If this property is defined, it is typically the name and email address of the feed editor. If an editor has not been specified, this property will return an empty string.

Data Type

String

See Also

Language Property, LinkUrl Property, Published Property, Title Property

FeedVersion Property

Gets a value which identifies the version of the news feed.

Syntax

object.**FeedVersion**

Remarks

The **FeedVersion** property returns a string that identifies the version of the news feed. This can be used by an application to determine which version of the RSS specification was used to create the feed. The current version of the specification is 2.0.

Data Type

String

See Also

Language Property, LinkUrl Property, Published Property, Title Property

Generator Property

Gets a value which identifies the application that was used to create the news feed.

Syntax

object.**Generator**

Remarks

The **Generator** property returns a string which identifies the application that was used to create the news feed. If the feed does not specify a generator, then this property will return an empty string.

Data Type

String

See Also

Description Property, Language Property, LinkUrl Property, Published Property, Title Property

ImageLink Property

Gets a value which specifies a URL to the website corresponding to the news feed.

Syntax

object.**ImageLink**

Remarks

The **ImageLink** property returns a string that specifies a URL to the website corresponding to the channel. In most cases, this is the same URL that is specified by the **LinkUrl** property. If an image link has not been specified, this property will return an empty string.

Data Type

String

See Also

ImageTitle Property, ImageUrl Property, LinkUrl Property

ImageTitle Property

Gets a value which describes the image associated with the news feed.

Syntax

object.**ImageTitle**

Remarks

The **ImageTitle** property returns a string that identifies the image associated with the channel. This is usually a brief description of the image, and may be the same as the value specified by the **Title** property. If an image title has not been specified, this property will return an empty string.

Data Type

String

See Also

ImageLink Property, ImageUrl Property, Title Property

ImageUrl Property

Gets a value which specifies a URL for the image associated with the news feed.

Syntax

object.**ImageUrl**

Remarks

The **ImageUrl** property returns a string that specifies a URL for the image associated with the channel. An application can download this image and display it with the contents of the news feed. If an image URL has not been specified, this property will return an empty string.

Data Type

String

See Also

ImageLink Property, ImageTitle Property

ItemAuthor Property

Gets a value which identifies the author of the current news feed item.

Syntax

object.**ItemAuthor**

Remarks

The **ItemAuthor** property returns a string which specifies an email address. If this property is defined, it is typically the name and address of the person who created the content that the item links to. If the author is not specified, this property will return an empty string.

Data Type

String

See Also

ItemGuid Property, ItemLink Property, ItemText Property, ItemTitle Property

ItemComments Property

Gets a value which specifies a URL that links to further discussion about the current item.

Syntax

object.**ItemComments**

Remarks

The **ItemComments** property returns a string which specifies a URL that links to further discussion about the item. Typically this is a link to the comment area of a weblog or a forum topic specific to the item. If a comment link is not specified, this property will return an empty string.

Data Type

String

See Also

ItemAuthor Property, ItemGuid Property, ItemLink Property, ItemText Property, ItemTitle Property

ItemCount Property

Gets value which specifies the number of news items in the channel.

Syntax

object.**ItemCount**

Remarks

The **ItemCount** property returns an integer value which specifies the number of items in the current news feed channel. This property can be used in conjunction with the **GetItem** method to enumerate through the available news feed items.

Data Type

Integer (Int32)

Example

The following example accesses a remote feed and enumerates each news item, populating the contents of a ListBox control with its title.

```
Dim strFeed As String
Dim nIndex As Long
Dim nError As Long

strFeed = "http://sockettools.com/rss/news.xml"
ListBox1.Clear

nError = NewsFeed1.Open(strFeed)
If nError > 0 Then
    MsgBox NewsFeed1.LastErrorString, vbExclamation
    Exit Sub
End If

Label1.Caption = NewsFeed1.ItemCount & " news items, published on " &
NewsFeed1.Published

For nIndex = 1 To NewsFeed1.ItemCount
    nError = NewsFeed1.GetItem(nIndex)
    If nError > 0 Then
        MsgBox NewsFeed1.LastErrorString, vbExclamation
        Exit For
    End If
    ListBox1.AddItem NewsFeed1.ItemTitle
Next
```

See Also

ItemGuid Property, ItemId Property, ItemText Property, GetItem Method

ItemEnclosure Property

Gets a value which specifies a URL that links to a file related to the item.

Syntax

object.**ItemEnclosure**

Remarks

The **ItemEnclosure** property returns a string which specifies a URL that links to an attached document for the news feed item. This is similar to an attachment in an email message, however instead of the item containing the contents of the attached file, it only specifies a link to the file. Enclosures are most commonly used with podcasting where an item is linked to an audio or video file, however the link may reference any type of file. If there is no enclosure specified for the current item, this property will return an empty string.

Data Type

String

See Also

ItemAuthor Property, ItemGuid Property, ItemLink Property, ItemText Property, ItemTitle Property

ItemGuid Property

Gets a value which uniquely identifies the current news item in the channel.

Syntax

object.**ItemGuid**

Remarks

The **ItemGuid** property returns a string which uniquely identifies the current news item. If this property is defined, it is guaranteed to be a unique, persistent value. It is important to note that this string does not have to be a standard GUID reference number, it can be any unique string. In many cases it is the same value as the hyperlink returned by the **ItemLink** property, although an application should never depend on this behavior. If there is no unique identifier associated with the current item, this property will return an empty string.

Data Type

String

See Also

ItemAuthor Property, ItemId Property, ItemLink Property, ItemText Property, ItemTitle Property

ItemId Property

Gets an numeric value which identifies the current news item in the channel.

Syntax

object.**ItemId**

Remarks

The **ItemId** property returns an integer that is an index into the list of available news items. It is not persistent and the ID for a specific news item may change when the news feed is refreshed or opened at a later point. To uniquely identify a news item in the channel, use the **ItemGuid** property. To retrieve a news feed item, use the **GetItem** method.

Data Type

Integer (Int32)

See Also

ItemAuthor Property, ItemGuid Property, ItemText Property, GetItem Method

ItemLink Property

Gets a value which specifies a URL that links to additional information related to the current item.

Syntax

object.**ItemLink**

Remarks

The **ItemLink** property returns a string which specifies a URL that provides additional information about a news item. If the news item summarizes the contents of an article, this property typically provides a link to the complete article. If a link is not specified, this property will return an empty string.

Data Type

String

See Also

ItemAuthor Property, ItemGuid Property, ItemText Property, ItemTitle Property

ItemPublished Property

Gets a value which specifies the date and time the current news item was published.

Syntax

object.**ItemPublished**

Remarks

The **ItemPublished** property returns a string which specifies the date and time that the news item was published. If the news item does not specify the publish date, this property will return an empty string. The date and time value returned is in the standard format used by the current locale.

Data Type

String

See Also

ItemAuthor Property, ItemGuid Property, ItemLink Property, ItemText Property, ItemTitle Property

ItemSource Property

Gets a value which identifies the source of the current news item.

Syntax

object.**ItemSource**

Remarks

The **ItemSource** property returns a string which specifies a URL for the original feed that contained the news item. This is typically used to propagate credit for news items that are aggregated by a third-party and re-published in their own channel. If the source is not specified, this property will return an empty string.

Data Type

String

See Also

ItemAuthor Property, ItemGuid Property, ItemLink Property, ItemText Property, ItemTitle Property

ItemText Property

Gets a value which provides a summary or description of the current news item.

Syntax

object.**ItemText**

Remarks

The **ItemText** property returns a string that contains a summary of the current news item. This may property may return either plain text or HTML formatted text. If no text has been specified for the current item, this property will return an empty string. Although it is not required for a news item to have a description, a feed that conforms to the standard must have either a description of the item or a title, which is returned by the **ItemTitle** property.

Data Type

String

See Also

ItemAuthor Property, ItemGuid Property, ItemLink Property, ItemTitle Property

ItemTitle Property

Gets a value which specifies the title for the current news item.

Syntax

object.**ItemTitle**

Remarks

The **ItemTitle** property returns a string which specifies a title for the news item. If no title has been specified, this property will return an empty string. Although it is not required for a news item to have a title, a feed that conforms to the standard must have either a title or a description of the item, which is returned by the **ItemText** property.

Data Type

String

See Also

ItemAuthor Property, ItemGuid Property, ItemLink Property, ItemText Property

Language Property

Gets a value which identifies the language the news feed is written in.

Syntax

object.**Language**

Remarks

The **Language** property returns a string which defines the language the channel is written in, using the standard language codes or an empty string if the language is not defined. This property typically returns standardized language codes, however the value actually returned depends on the content of the feed. If the news feed does not define this property, then it is generally presumed to be written in English.

Data Type

String

See Also

Description Property, Language Property, LinkUrl Property, Published Property, Title Property

LastBuild Property

Gets a value which specifies the date and time that the content of the news feed was last modified.

Syntax

object.**LastBuild**

Remarks

The **LastBuild** property returns a string which specifies the date and time that the feed was last modified. If the feed does not define this value, this property will return an empty string. The date and time value returned is in the standard format used by the current locale.

Data Type

String

See Also

Description Property, LinkUrl Property, Published Property, Title Property

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 to occur

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, ThrowError Property, OnError Event

LinkUrl Property

Gets a value which specifies a URL to the website corresponding to the channel.

Syntax

object.**LinkUrl**

Remarks

The **LinkUrl** property returns a string which specifies a URL to the website corresponding to the channel. Note that this is not the URL of the news feed itself. Typically it is a link to the home page of the site which owns the news feed. If a link has not been specified, this property will return an empty string. Note that a strictly conforming news feed requires a valid link URL.

Data Type

String

See Also

Description Property, LastBuild Property, Published Property, Title Property

LocalFeed Property

Gets a value which specifies if the news feed was opened on the local system.

Syntax

object.**LocalFeed**

Remarks

This property will return true if the news feed was accessed from the local system by specifying a file name to the **Open** method. If the news feed was accessed from a remote web server, this property will return false.

Data Type

Boolean

See Also

Url Property, Open Method

Published Property

Gets a value which specifies the date and time that the news feed was published.

Syntax

object.**Published**

Remarks

The **Published** property returns a string which specifies the date and time that the feed was published. For example, a feed that is associated with a weekly print publication may update this value once per week. Note that this is not necessarily the date that the feed was last modified. If the feed channel does not specify the publish date, this property will return an empty string. The date and time value returned is in the standard format used by the current locale.

Data Type

String

See Also

Description Property, LastBuild Property, LinkUrl Property, Title Property

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
NewsFeed1.ThrowError = False
nError = NewsFeed1.Open(strFeedUrl)

If nError > 0 Then
    MsgBox NewsFeed1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

NewsFeed1.ThrowError = True
NewsFeed1.Open strHostName

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error.

Note that the **Timeout** property also determines the amount of time the control will spend attempting to connect to a server. If a connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

TimeToLive Property

Gets a value which specifies the frequency in seconds at which the feed should be refreshed.

Syntax

object.**TimeToLive**

Remarks

The **TimeToLive** property is an integer value that specifies the frequency in seconds at which the feed should be refreshed to obtain updated information. Not all feeds specify a time-to-live, in which case this property will have a value of zero.

The value of the **TimeToLive** property should be considered advisory, and not all news feeds will provide this value. If the news feed does provide this value, it is recommended that you consider it to be the minimum interval at which you will poll the site for updates to the feed.

Data Type

Integer (Int32)

See Also

Open Method, Refresh Method

Title Property

Gets a value which specifies the name of the news feed channel.

Syntax

object.**Title**

Remarks

The **Title** property returns a string which specifies the title of the news feed channel. If the content of the news feed corresponds to a website, this is value returned by this property is typically the same as the title of the website. If a title has not been specified, this property will return an empty string. Note that a strictly conforming news feed requires a title.

Data Type

String

See Also

Description Property, LastBuild Property, Published Property

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
controlTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
controlTraceError	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.
controlTraceWarning	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.
controlTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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

URL Property

Gets and sets a value which specifies the news feed URL.

Syntax

object.**URL** [= *url*]

Remarks

The **URL** property returns a string which specifies a URL to the news feed. This may be either an http:// or https:// URL to specify a news feed on a web server, or it may be a file:// URL that specifies a local XML file that contains the news feed.

Data Type

String

See Also

Description Property, LastBuild Property, Published Property, Title Property

Version Property

Return the current version of the object.

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

See Also

FeedVersion Property

Webmaster Property

Gets a value which identifies the person responsible for technical issues related to the news feed.

Syntax

object.**Webmaster**

Remarks

The **Webmaster** property returns a string which contains an email address. If this value is defined in the news feed, it is typically the address of a system administrator responsible for the server that hosts the news feed. If the webmaster is not specified, this property will return an empty string.

Data Type

String

See Also

Description Property, Language Property, LinkUrl Property, Published Property, Title Property

News Feed Control Methods

Method	Description
Close	Close the current news feed
FindItem	Search for an item in the news feed channel which matches the unique identifier
GetItem	Set the current news item to the specified item number
GetProperty	Get the value of a property for the specified item in the news feed
Initialize	Initialize the control and validate the runtime license key
Open	Open the specified news feed and load the first news item
Parse	Parse the contents of a news feed and load the first news item
Refresh	Refresh the current news feed, reloading the news channel items
Reset	Reset the internal state of the control
Store	Store the contents of the news feed in an XML formatted text file
Uninitialize	Uninitialize the control and release any system resources that were allocated

Close Method

Close the current news feed.

Syntax

object.**Close**

Parameters

None.

Return Value

A value of zero is returned if the feed was closed successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure. If the method fails, the value of the **LastError** property can be used to determine cause of the failure.

Remarks

The **Close** method must be called whenever the application has completed processing the news feed. Information about the current news feed item will be cleared whenever this method is called, resetting the channel and item related properties back to their default values.

See Also

FindItem Method, GetItem Method, Open Method, Parse Method, Store Method

FindItem Method

Search for an item in the news feed channel which matches the unique identifier.

Syntax

object.**FindItem**(*Value*, [*Options*])

Parameters

Value

A string which specifies the value of the news item being searched for. This value should uniquely identify the item in the feed, and this parameter cannot be an empty string.

Options

An optional numeric parameter that specifies one or more options when searching for a news item. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
rssFindGuid	Search the feed for an item with a matching GUID property value. This is the default option, and is the only item property that is guaranteed to be unique in the feed. The search is case-sensitive, requiring that the ***Value*** parameter match the GUID property value for the news item exactly.
rssFindLink	Search the feed for items with a matching link property value. For feeds that do not specify a GUID property, this is the recommended option for searching for an item. The search is not case-sensitive.
rssFindTitle	Search the feed for items with a matching title. This option should not be used if you must ensure that the item returned is unique in the feed because there may be multiple items with the same title in the feed. The search is not case-sensitive.
rssFindPubDate	Search the feed for items with a matching publishing date. This option should not be used if you must ensure that the item returned is unique in the feed because more than one item may have the same publishing date. The format of the date string must match the standard format used with the RSS protocol and the match is not case-sensitive.

Return Value

A value of zero is returned if the news item was found. Otherwise, a non-zero error code is returned which indicates the cause of the failure. If the method fails, the value of the **LastError** property can be used to determine cause of the failure.

Remarks

The **FindItem** method searches for an item in the news feed channel which matches the unique identifier (GUID) value and returns information about that item. If this method is successful, the current news item is changed to the item that was found and property values such as **ItemLink** and **ItemText** will be updated. If this method fails, the current news item is not changed.

It is recommended that you use this method with news feeds that are using version 2.0 or later of the RSS specification. If the feed uses an earlier version, items may not include a GUID property. It is also possible that a feed may omit the GUID property even though it is considered a requirement for the current RSS specification. For the broadest compatibility with all news feeds, an application should not

depend on being able to search for a specific news feed item by its GUID.

See Also

Close Method, GetItem Method, Open Method, Parse Method

GetItem Method

Set the current news item to the specified item number.

Syntax

object.**GetItem**(*ItemId*)

Parameters

ItemId

An integer value which specifies item in the news feed channel.

Return Value

A value of zero is returned if the news item was selected. Otherwise, a non-zero error code is returned which indicates the cause of the failure. If the method fails, the value of the **LastError** property can be used to determine cause of the failure.

Remarks

The **GetItem** method is used to select the current news item in the feed. If this method is successful, the current news item is changed to the specified value and property values such as **ItemLink** and **ItemText** will be updated. If this method fails, the current news item is not changed.

The item number is an index into the list of available news items in the current news feed. The first news item is one, and it increments for each additional item in the feed. If *ItemId* parameter is zero or specifies a value larger than the number of items in the feed, this method will fail. The **ItemId** property returns the value of the currently selected news item.

If this method fails, it typically indicates that the *ItemId* parameter is invalid or that the feed does not contain any valid news items. The **ItemCount** property can be used to determine the number of items contained in the news feed channel.

Example

The following example accesses a remote feed and enumerates each news item, populating the contents of a ListBox control with its title.

```
Dim strFeed As String
Dim nIndex As Long
Dim nError As Long

strFeed = "http://sockettools.com/rss/news.xml"
ListBox1.Clear

nError = NewsFeed1.Open(strFeed)
If nError > 0 Then
    MsgBox NewsFeed1.LastErrorString, vbExclamation
    Exit Sub
End If

Label1.Caption = NewsFeed1.ItemCount & " news items, published on " &
NewsFeed1.Published

For nIndex = 1 To NewsFeed1.ItemCount
    nError = NewsFeed1.GetItem(nIndex)
    If nError > 0 Then
        MsgBox NewsFeed1.LastErrorString, vbExclamation
        Exit For
```

```
        End If
        ListBox1.AddItem NewsFeed1.ItemTitle
    Next
```

See Also

ItemCount Property, ItemId Property, Close Method, FindItem Method, Open Method, Parse Method

GetProperty Method

Get the value of a property for the current item in the news feed.

Syntax

object.**GetProperty**(*Property*, *Value*)

Parameters

Property

A string which specifies the name of the custom property to retrieve the value for.

Value

A string that is passed by reference which will contain the property value when the method returns.

Return Value

A value of zero is returned if the property was specified in the news item. Otherwise, a non-zero error code is returned which indicates the cause of the failure. If the method fails, the value of the **LastError** property can be used to determine cause of the failure.

Remarks

The **GetProperty** method is primarily used with custom item properties that may be used with extensions to the news feed. The standard properties for an news feed item such as the title, link and description can be access using properties such as **ItemTitle**, **ItemLink** and **ItemText**. However, if items in the feed contain custom properties that are not part of the standard RSS format, this method can be used to obtain those values.

See Also

Close Method, GetItem Method, Open Method, Parse 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 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 rssFeed = CreateObject("SocketTools.NewsFeed.11")

nError = rssFeed.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Open Method, Parse Method, Uninitialize Method

Open Method

Open the specified news feed and select the first news item.

Syntax

object.**Open**(*FeedUrl*, [*Timeout*], [*Options*])

Parameters

FeedUrl

A string value which specifies the URL for the news feed. To access a news feed on a web server, a standard http or https URL may be used. To access a file on the local system or network share, a file name or file URL may be specified.

Timeout

The number of seconds that the client will wait for a response before failing the operation.This parameter is ignored if the ***FeedUrl*** parameter specifies a local file name or URL.

Options

An optional numeric parameter that specifies one or more options when opening the news feed. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
rssOptionNone	No additional options are specified and the news feed is processed using relaxed rules when checking the validity of the feed. The control will attempt to automatically compensate for a feed that is malformed or does not strictly conform to the RSS standard. This is the default value if the ***Options*** parameter is omitted.
rssOptionStrict	The news feed content should be processed using strict rules to ensure that the feed meets the appropriate RSS standard specification and all feed property values are case-sensitive. By default, relaxed rules are used which allows the application to open a feed that may not strictly conform to the standard specification.

Return Value

A value of zero is returned if the news feed was opened successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure. If the method fails, the value of the **LastError** property can be used to determine cause of the failure.

Remarks

A news feed may be local or remote, depending on the URL that is specified. If a local file name or file URL is specified for the feed, then it is opened locally and no network access is required. If an http or https URL is specified, then the **Open** method will attempt to download the feed from the server and store it temporarily on the local system. Accessing a remote feed requires that the application has permission to establish a connection with the server and will cause the application to block until the feed has been downloaded, the operation times out or an error occurs.

Although the **Open** method will meet the needs of most applications, if you require more complex functionality such as retrieving the feed asynchronously in the background or event notifications for large transfers, you can use the SocketTools Hypertext Transfer Protocol control to download the news feed and then use the **Parse** method to parse the contents.

Example

The following example accesses a remote feed and enumerates each news item, populating the contents of a ListBox control with its title.

```
Dim strFeed As String
Dim nIndex As Long
Dim nError As Long

strFeed = "http://sockettools.com/rss/news.xml"
ListBox1.Clear

nError = NewsFeed1.Open(strFeed)
If nError > 0 Then
    MsgBox NewsFeed1.LastErrorString, vbExclamation
    Exit Sub
End If

Label1.Caption = NewsFeed1.ItemCount & " news items, published on " &
NewsFeed1.Published

For nIndex = 1 To NewsFeed1.ItemCount
    nError = NewsFeed1.GetItem(nIndex)
    If nError > 0 Then
        MsgBox NewsFeed1.LastErrorString, vbExclamation
        Exit For
    End If
    ListBox1.AddItem NewsFeed1.ItemTitle
Next
```

See Also

Close Method, GetItem Method, Open Method, Parse Method, Store Method

Parse Method

Open the specified news feed and select the first news item.

Syntax

object.**Parse**(*FeedXml*, [*Options*])

Parameters

FeedXml

A string value which contains the contents of the news feed to be parsed.

Options

An optional numeric parameter that specifies one or more options when opening the news feed. This parameter is constructed by using a bitwise operator with any of the following values:

Value	Description
rssOptionNone	No additional options are specified and the news feed is processed using relaxed rules when checking the validity of the feed. The control will attempt to automatically compensate for a feed that is malformed or does not strictly conform to the RSS standard. This is the default value if the *Options* parameter is omitted.
rssOptionStrict	The news feed content should be processed using strict rules to ensure that the feed meets the appropriate RSS standard specification and all feed property values are case-sensitive. By default, relaxed rules are used which allows the application to open a feed that may not strictly conform to the standard specification.

Return Value

A value of zero is returned if the string contains a valid RSS feed and the contents were successfully parsed. Otherwise, a non-zero error code is returned which indicates the cause of the failure. If the method fails, the value of the **LastError** property can be used to determine cause of the failure.

Remarks

The **Parse** method is an alternative to the **Open** method, enabling the application to process a news feed from alternative sources such as a database or compressed file. It is important to note that the string which contains the news feed XML must be properly formatted and conform to the RSS standard specification.

Example

The following example opens a local file that contains a news feed, stores the contents in a string variable and parses the contents. A **ListBox** control is populated with the title of each news item in the feed. Note that this example was written to demonstrate the use of the **Parse** method, however the **Open** method can also be used to open a local file and requires less code.

```
Dim hFile As Long
Dim strFileName As String
Dim strFeedXml As String
Dim nIndex As Long
Dim nError As Long

strFileName = "newsfeed.xml"
```

```
hFile = FreeFile()
Open strFileName For Input As #hFile
strFeedXml = Input(LOF(hFile), #hFile)
Close #hFile

nError = NewsFeed1.Parse(strFeedXml)
If nError > 0 Then
    MsgBox NewsFeed1.LastErrorString, vbExclamation
    Exit Sub
End If

ListBox1.Clear
Label1.Caption = NewsFeed1.ItemCount & " news items, published on " &
NewsFeed1.Published

For nIndex = 1 To NewsFeed1.ItemCount
    nError = NewsFeed1.GetItem(nIndex)
    If nError > 0 Then
        MsgBox NewsFeed1.LastErrorString, vbExclamation
        Exit For
    End If
    ListBox1.AddItem NewsFeed1.ItemTitle
Next
```

See Also

Close Method, GetItem Method, Open Method, Store Method

Refresh Method

Refresh the current news feed.

Syntax

object.**Refresh**

Parameters

None.

Return Value

A value of zero is returned if the feed was refreshed successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure. If the method fails, the value of the **LastError** property can be used to determine cause of the failure.

Remarks

When the **Refresh** method is called, the news feed is reloaded from the original source and the items in the channel are updated. For news feeds that are frequently updated, the **TimeToLive** property can provide a hint to the application as to how frequently the feed should be refreshed.

If the news feed was originally opened using an http or https URL, this function will download the updated feed from the server and store it temporarily on the local system. Accessing a remote feed requires that the application has permission to establish a connection with the server and will cause the application to block until the feed has been downloaded, the operation times out or an error occurs. The same timeout period and options will be used as when the feed was originally opened.

The **Refresh** method should only be used if the feed was opened using the **Open** method, otherwise the method will fail with an error indicating that the operation is not supported.

It is important that the application does not make any assumptions about the number of news items in the channel, or the content associated with those items after the **Refresh** method has been called. For example, never assume that the number of items in the news feed remains the same, or that the item IDs for each item remains the same. If you need to find a specific item in the news feed, use the **FindItem** method.

See Also

TimeToLive Property, FindItem Method, GetItem Method, Open Method, Parse 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, open network connections will be closed and any handles allocated by the control will be released.

See Also

Initialize Method, Uninitialize Method

Store Method

Store the contents of the news feed in an XML formatted text file.

Syntax

object.**Store**(*FileName*, [*Options*])

Parameters

FileName

A string value which specifies the name of the file on the local system. The contents of the news feed will be stored in this file. If the file does not exist, it will be created; otherwise it will overwrite the contents of the file.

Options

An optional parameter reserved for future use. It should either be omitted from the method call, or passed with a value of 0.

Return Value

A value of zero is returned if the contents of the news feed has been successfully stored in the specified file. Otherwise, a non-zero error code is returned which indicates the cause of the failure. If the method fails, the value of the **LastError** property can be used to determine cause of the failure.

See Also

Close Method, GetItem Method, Open Method, Parse 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. 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

Close Method, Initialize Method, Reset Method

News Feed Control Events

Event	Description
OnError	This event is generated when a control error occurs

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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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

Network News Transfer Protocol Control

Download and submit articles to a news server.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	NntpClientCtl.NntpClient
File Name	CSNWSX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.NntpClient.11
ClassID	DEC81A71-4F58-4E03-B601-E486822DBD1F
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 977, RFC 2980

Overview

The Network News Transfer Protocol (NNTP) is used with servers that provide news services. This is similar in functionality to bulletin boards or message boards, where topics are organized hierarchically into groups, called newsgroups. Users can browse and search for messages, called news articles, which have been posted by other users. On many servers, they can also post their own articles which can be read by others. The largest collection of public newsgroups available is called USENET, a world-wide distributed discussion system. In addition, there are a large number of smaller news servers. For example, Microsoft operates a news server which functions as a forum for technical questions and announcements.

The control provides a comprehensive interface for accessing newsgroups, retrieving articles and posting new articles. In combination with the Mail Message control to process the news articles, SocketTools can be used to integrate newsgroup access with an existing email application, or you can implement your own full-featured newsgroup client.

This control supports secure connections using the standard SSL and TLS protocols.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a

minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

Network News Transfer Protocol Control Properties

Property	Description
Article	Gets and sets the current article number
ArticleCount	Return the number of available articles
AutoResolve	Determines if host names and IP addresses are automatically resolved
Blocking	Gets and sets the blocking state of the control
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 client certificate
CertificatePassword	Gets and sets the password associated with the client certificate
CertificateStatus	Return the status of the server certificate
CertificateStore	Gets and sets the name of the client 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
CurrentDate	Return the current date in the standard format used by news articles
FirstArticle	Return the first available article number
GroupCount	Return the number of available groups
GroupName	Gets and sets the current newsgroup name
GroupTitle	Return a string describing the current newsgroup
HashStrength	Return the length of the message digest that was selected
HostAddress	Gets and sets the IP address of the server
HostName	Gets and sets the name of the server
IsBlocked	Return if the control is blocked performing an operation
IsConnected	Determine if the control is connected to a server
IsInitialized	Determine if the control has been initialized
IsReadable	Return if data can be read from the server without blocking
IsWritable	Return if data can be sent to the server without blocking
LastArticle	Return the last available article number
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error to occur
LastUpdate	Gets and sets the date the newsgroup list was last updated
MessageId	Gets and sets the current article by message ID
Options	Gets and sets the options that are used in establishing a connection

Password	Gets and sets the password for the current user
RemotePort	Gets and sets the port number for a remote connection
ResultCode	Return the result code of the previous action
ResultString	Return a string describing the results of the previous action
Secure	Set or return if a connection to the server is secure
SecureCipher	Return the encryption algorithm used to establish the secure connection with the server
SecureHash	Return the message digest selected when establishing the secure connection with the server
SecureKeyExchange	Return the key exchange algorithm used to establish the secure connection with the server
SecureProtocol	Gets and sets the security protocol used to establish the secure connection with the server
ThrowError	Enable or disable error handling by the container of the control
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
UserName	Gets and sets the current user name
Version	Return the current version of the object

Article Property

Gets and sets the current article number.

Syntax

object.**Article** [= *number*]

Remarks

The Article property sets or returns the current news article number. Setting the Article property updates the MessageId property to reflect the specified article's message ID.

Data Type

Integer (Int32)

See Also

ArticleCount Property, FirstArticle Property, LastArticle Property, MessageId Property

ArticleCount Property

Return the number of available articles in the current newsgroup.

Syntax

object.**ArticleCount**

Remarks

The **ArticleCount** property returns the number of articles that are available in the current newsgroup. Note that this is a read-only property available at run-time. This value is only meaningful after the **SelectGroup** method has been called.

Data Type

Integer (Int32)

See Also

Article Property, FirstArticle Property, LastArticle Property, ListArticles Method, SelectGroup Method

AutoResolve Property

Determines if host names and IP 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 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 **AutoResolve** to False.

Data Type

Boolean

See Also

HostAddress Property, HostName Property

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 read data when none has been written, an error is generated. Events such as **OnConnect**, **OnDisconnect**, **OnRead** and **OnWrite** are only fired if the connection is non-blocking.

Data Type

Boolean

See Also

IsBlocked Property, IsReadable Property, IsWritable Property

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 = NntpClient1.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 client certificate.

Syntax

object.**CertificateName** [= *name*]

Remarks

This property sets the common name or friendly name of the certificate that should be used to establish the connection with the server. 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. If a certificate name is specified, the certificate must have a private key associated with it, otherwise the connection attempt 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 client 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
stCertificateNone	No certificate information is available. A secure connection was not established with the server.
stCertificateValid	The certificate is valid.
stCertificateNoMatch	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.
stCertificateExpired	The certificate has expired and is no longer valid. The client can examine the **CertificateExpires** property to determine when the certificate expired.
stCertificateRevoked	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.
stCertificateUntrusted	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.
stCertificateInvalid	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:

```
'
' Initialize the control properties
```

```
'

NntpClient1.HostName = strHostName
NntpClient1.Secure = True

nError = NntpClient1.Connect()
If nError > 0 Then
     MsgBox "Unable to connect to server " & strHostName, vbExclamation
     Exit Sub
End If

If NntpClient1.CertificateStatus <> stCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          NntpClient1.Disconnect
          Exit Sub
     End If
End If

NntpClient1.Disconnect
```

See Also

CertificateExpires Property, CertificateIssued Property, CertificateIssuer Property, CertificateSubject Property, Secure Property

CertificateStore Property

Gets and sets the name of the client certificate store or file.

Syntax

object.**CertificateStore** [= *store*]

Remarks

This property sets the name of the certificate store that contains the client certificate that should be used when establishing a secure connection with 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 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

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 = NntpClient1.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 client certificate.

Syntax

object.**CertificateUser** [= *username*]

Remarks

This property sets the name of the user that owns the client certificate that will be used to establish a secure connection with the server. 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

CurrentDate Property

Return the current date in the standard format used by news articles.

Syntax

object.**CurrentDate**

Remarks

The **CurrentDate** property returns the current date and time in a format that is commonly used in news articles.

Data Type

String

FirstArticle Property

Return the first available article number.

Syntax

object.**FirstArticle**

Remarks

The **FirstArticle** property returns the first article number available in the current newsgroup. Note that this is a read-only property available at run-time.

Data Type

Integer (Int32)

See Also

Article Property, ArticleCount Property, LastArticle Property

GroupCount Property

Return the number of available groups.

Syntax

object.**GroupCount**

Remarks

The **GroupCount** property returns the number of newsgroups that have been returned by the server. Note that this is a read-only property available at run-time and the value is only meaningful after the **ListGroups** method has been called.

Data Type

Integer (Int32)

See Also

GroupName Property, GroupTitle Property, ListGroups Method

GroupName Property

Gets and sets the current newsgroup name.

Syntax

object.**GroupName** [= *group*]

Remarks

The **GroupName** property sets or returns the current newsgroup. Setting this property causes the related properties, such as **FirstArticle** and **LastArticle**, to be updated.

Data Type

String

See Also

GroupCount Property, GroupTitle Property

GroupTitle Property

Return a string describing the current newsgroup.

Syntax

object.**GroupTitle**

Remarks

The **GroupTitle** property returns a string describing the current newsgroup. This property is read-only and available only at run-time.

Data Type

String

See Also

GroupCount Property, GroupName Property

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 server.

Syntax

object.**HostAddress** [= *ipaddress*]

Remarks

The **HostAddress** property can be used to set the IP address for a server 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, HostName Property

HostName Property

Gets and sets the name of the server.

Syntax

object.**HostName** [= *hostname*]

Remarks

The **HostName** property should be set to the name of the server 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

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.

Note that this property will return True if there is ***any*** blocking operation being performed by the application, regardless if the specified control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

Blocking Property, LastError Property

IsConnected Property

Determine if the control is connected to a server.

Syntax

object.**IsConnected**

Remarks

The **IsConnected** read-only property is set to a value of true if the control is connected with a server, otherwise the property has a value of false.

Data Type

Boolean

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

IsReadable Property

Return if data can be read from the server without blocking.

Syntax

object.**IsReadable**

Remarks

The **IsReadable** property returns True if data can be read from the server without blocking. For non-blocking connections, this property can be checked before the application attempts to read the data, preventing an error.

Data Type

Boolean

See Also

IsConnected Property, Read Method, OnRead Event

IsWritable Property

Return if data can be sent to the server without blocking.

Syntax

object.**IsWritable**

Remarks

The **IsWritable** property returns True if data can be written without blocking. For non-blocking connections, this property can be checked before the application attempts to send data to the server, 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 send data without an error. The next operation may result in an **stErrorOperationWouldBlock** or **stErrorOperationInProgress** error. The application must treat these errors as recoverable, and should be prepared to retry operations that result in them.

Data Type

Boolean

See Also

IsReadable Property, Write Method, OnWrite Event

LastArticle Property

Return the last available article number.

Syntax

object.**LastArticle**

Remarks

The **LastArticle** property returns the last article number available in the current newsgroup. Note that this is a read-only property available at run-time.

Data Type

Integer (Int32)

See Also

Article Property, ArticleCount Property, FirstArticle Property

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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

LastUpdate Property

Gets and sets the date the newsgroup list was last updated.

Syntax

object.**LastUpdate** [= *date*]

Remarks

The **LastUpdate** property is used to specify the last date and time that a list of newsgroups was retrieved from the server. When this property is set, only those newsgroups created after that date will be returned when the server is asked for a list of groups. If this property is set to an empty string, then the server will return all groups, regardless of when they were created.

This property should be used by client applications to reduce the amount of time needed to update the list of available newsgroups, particularly if the server offers a large number of newsgroups. For example, the first time that the client connects to the server, the **LastUpdate** property should not be set. That will cause the server to return a list of all of the newsgroups that it has available. The client should then store that list on the local system in a file, and record the date and time that it created the list. Then, the next time that the client connects to the server, it should set the **LastUpdate** property to the date that the local list of newsgroups was last updated. When the list of newsgroups is requested again, the server will only return those newsgroups that were created since the last time the list was updated rather than the complete list.

Data Type

String

See Also

GetFirstGroup Method, GetNextGroup Method, ListGroups Method, OnNewsGroup Event

MessageID Property

Gets and sets the current article by message ID.

Syntax

object.**MessageId** [= *number*]

Remarks

The **MessageId** property sets or returns the current message ID string. Each news article has a unique string which identifies that message. Setting the **MessageId** property causes the current article number to change to the given message. An error is generated if the property is set to an invalid message ID.

Data Type

String

See Also

Article Property, ArticleCount Property, FirstArticle Property, LastArticle Property

Options Property

Gets and sets the options that are used in establishing a connection.

Syntax

object.**Options** [= *value*]

Remarks

The **Options** property is an integer value which specifies one or more options. The value specified for this property will be used as the default options when connecting to the server. The property value is created by using a bitwise operator with one or more of the following values:

Value	Description	
nntpOptionNone	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used.	
&H400	nntpOptionTunnel	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
&H800	nntpOptionTrustedSite	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 the TLS protocol.
&H1000	nntpOptionSecure	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
&H8000	nntpOptionSecureFallback	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.
&H40000	nntpOptionPreferIPv6	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.

Data Type

Integer (Int32)

See Also

Secure Property, Connect Method

Password Property

Gets and sets the password for the current user.

Syntax

object.**Password** [= *password*]

Remarks

The **Password** property specifies the password used to authenticate the user. If a password is not required by the server, this property is ignored.

Data Type

String

See Also

UserName Property, Authenticate Method

RemotePort Property

Gets and sets the port number for a remote connection.

Syntax

object.**RemotePort** [= *portnumber*]

Remarks

The **RemotePort** property is used to set the port number that the control will use to establish a connection with the server.

Data Type

Integer (Int32)

See Also

HostAddress Property, HostName Property

ResultCode Property

Return the result code of the previous action.

Syntax

object.**ResultCode**

Remarks

The **ResultCode** read-only property returns the result code of the last action performed by the client. This property should be checked after the **Command** method is used to execute a command on the server to determine if the operation was successful. Result codes are three-digit numeric values returned by the server and may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

It is important to note that while some result codes have become standardized, not all servers respond to commands using the same result codes. For example, one server may respond with a result code of 221 to indicate success, while another may respond with a value of 235. It is recommended that applications check for ranges of values to determine if a command was successful, not a specific value.

Data Type

Integer (Int32)

See Also

ResultString Property, Command Method, OnCommand Event

ResultString Property

Return a string describing the results of the previous action.

Syntax

object.**ResultString**

Remarks

The **ResultString** read-only property returns the result string from the last action taken by the client. This string is generated by the server, and typically is used to describe the result code. For example, if an error is indicated by the result code, the result string may describe the condition that caused the error.

Data Type

String

See Also

ResultCode Property, Command Method, OnCommand Event

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 must 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 strongly recommended that any application that sets this property True use error handling to trap an errors that may occur. If the control 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.

Data Type

Boolean

Example

The following example establishes a secure connection to a server:

```
NntpClient1.HostName = strHostName
NntpClient1.UserName = strUserName
NntpClient1.Password = strPassword
NntpClient1.Secure = True

nError = NntpClient1.Connect()
If nError > 0 Then
    MsgBox "Unable to connect to server " & strHostName, vbExclamation
    Exit Sub
End If

If NntpClient1.CertificateStatus <> stCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          NntpClient1.Disconnect
          Exit Sub
     End If
End If
```

See Also

CertificateStatus 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
stCipherNone	No cipher has been selected. This is not a secure connection with the server.
stCipherRC2	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.
stCipherRC4	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.
stCipherRC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
stCipherDES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher using 56-bit keys.
stCipherDES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively using a 168-bit key length.
stCipherDESX	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.
stCipherAES	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.
stCipherSkipjack	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
stCipherBlowfish	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
stHashMD5	The MD5 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA1	The SHA-1 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA256	The SHA-256 algorithm has been selected.
stHashSHA384	The SHA-384 algorithm has been selected.
stHashSHA512	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
stKeyExchangeNone	No key exchange algorithm has been selected. This is not a secure connection with the server.
stKeyExchangeRSA	The RSA public key exchange algorithm has been selected.
stKeyExchangeKEA	The KEA public key exchange algorithm has been selected. This is an improved version of the Diffie-Hellman public key algorithm.
stKeyExchangeDH	The Diffie-Hellman public key exchange algorithm has been selected.
stKeyExchangeECDH	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

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
NntpClient1.ThrowError = False
nError = NntpClient1.Connect(strHostName)

If nError > 0 Then
    MsgBox NntpClient1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

NntpClient1.ThrowError = True
NntpClient1.Connect strHostName

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error.

Note that the **Timeout** property also determines the amount of time the control will spend attempting to connect to a server. If a connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
nntpTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
nntpTraceError	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.
nntpTraceWarning	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.
nntpTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the nntps being used. If multiple nntps have tracing enabled, the **TraceFlags** property should be set to the same value for each nntp. Changing the trace flags for any one instance of the nntp will affect the logging performed for all nntps 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

UserName Property

Gets and sets the current user name.

Syntax

object.**UserName** [= *username*]

Remarks

The **UserName** property specifies the user that is logging in to the server, and is required for authentication purposes.

Data Type

String

See Also

Password Property, Authenticate Method

Version Property

Return the current version of the object.

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

Network News Transfer Protocol Control Methods

Method	Description
Authenticate	Authenticate the client session
Cancel	Cancels the current blocking network operation
CloseArticle	Closes the current article that has been opened or created
Command	Send a custom command to the server
Connect	Establish a connection with a server
CreateArticle	Creates a new article in the current newsgroup
Disconnect	Terminate the connection with a server
GetArticle	Retrieve a article from the server
GetFirstArticle	Return the first available article in the currently selected newsgroup
GetFirstGroup	Return the first available newsgroup from the server
GetHeaders	Retrieves the headers for the specified article from the server
GetNextArticle	Return the next available article in the currently selected newsgroup
GetNextGroup	Return the next available newsgroup from the server
Initialize	Initialize the control and validate the runtime license key
ListArticles	Return a list of articles in the current newsgroup
ListGroups	Return a list of newsgroups available on the server
OpenArticle	Opens the specified article in the currently selected newsgroup
PostArticle	Post a new article to the current newsgroup
Read	Return data read from the server
Reset	Reset the internal state of the control
SelectGroup	Selects the specified newsgroup as the current newsgroup
StoreArticle	Retrieve an article from the current newsgroup and store it in a local file
Uninitialize	Uninitialize the control and release any system resources that were allocated
Write	Write data to the server

Authenticate Method

Authenticate the client session.

Syntax

object.**Authenticate**([*UserName*], [*Password*])

Parameters

UserName

An optional string argument which specifies the username used to authenticate the client session. If the argument is omitted, the value assigned to the **UserName** property will be used instead.

Password

An optional string argument which specifies the password used to authenticate the client session. If the argument is omitted, the value assigned to the **Password** property will be used instead.

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 **Authenticate** method identifies the client to the news server, which may be required to access certain newsgroups or to post articles. If the user name or password is invalid, an error will occur. This method should only be used if the server indicates that authentication is required by returning an error.

See Also

Password Property, UserName Property, Connect 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.

CloseArticle Method

Closes the current article that has been opened or created.

Syntax

object.**CloseArticle**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CloseArticle** method closes the current article that has been opened or created. If an article is being created, this function actually submits the article to the server. Note that the client application is responsible for generating the message headers as well as the body of the message. News articles conform to the same general characteristics of an email message.

See Also

CreateArticle Method, OpenArticle Method, Read Method, Write Method

Command Method

Send a custom command to the server.

Syntax

object.**Command**(*Command*, [*Parameters*], [*Options*])

Parameters

Command

A string which specifies the command to send. Valid commands vary based on the Internet protocol and the type of server that the client is connected to. Consult the protocol standard and/or the technical reference documentation for the server to determine what commands may be issued by a client application.

Parameters

An optional string which specifies one or more parameters to be sent along with the command. If more than one parameter is required, most Internet protocols require that they be separated by a single space character. Consult the protocol standard and/or technical reference documentation for the server to determine what parameters should be provided when issuing a specific command. If no parameters are required for the command, this argument may be omitted.

Options

A numeric value which specifies one or more options. If this argument is omitted, no command options will be used. The following values may be specified:

Value	Description
nntpCommandMultiLine	This option specifies the command will return multiple lines of data. Unlike a single line response, which consists of a result code and result string, a multi-line response consists of one or more lines of text, terminated by a special end-of-data marker.

Return Value

A value of zero is returned if the command was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure. To determine the result code returned by the server in response to the command, read the value of the **ResultCode** property.

Remarks

The **Command** method sends a command to the server and processes the result code sent back in response to that command. This method can be used to send custom commands to a server to take advantage of features or capabilities that may not be supported internally by the control.

See Also

ResultCode Property, ResultString Property, OnCommand Event

Connect Method

Establish a connection with a server.

Syntax

object.**Connect**([*RemoteHost*], [*RemotePort*], [*UserName*], [*Password*], [*Timeout*], [*Options*])

Parameters

RemoteHost

A string which specifies the host name or IP address of the server. If this argument is not specified, it defaults to the value of the **HostAddress** property if it is defined. Otherwise, it defaults to the value of the **HostName** property.

RemotePort

A number which specifies the port to connect to on the server. If this argument is not specified, it defaults to the value of the **RemotePort** property. A value of zero indicates that the default port number for this service should be used to establish the connection.

UserName

A string which specifies the name of the user used to authenticate access to the server. If this argument is not specified, it defaults to the value of the **UserName** property.

Password

A string which specifies the password used to authenticate the user. If this argument is not specified, it defaults to the value of the **Password** property.

Timeout

The number of seconds that the client will wait for a response before failing the operation. If this argument is not specified, the value of the **Timeout** property will be used as the default.

Options

A numeric value which specifies one or more options. If this argument is omitted or a value of zero is specified, a default, standard connection will be established. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description	
nntpOptionNone	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used.	
&H400	nntpOptionTunnel	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
&H800	nntpOptionTrustedSite	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 the TLS protocol.
&H1000	nntpOptionSecure	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
&H8000	nntpOptionSecureFallback	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.
&H40000	nntpOptionPreferIPv6	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.

See Also

HostAddress Property, HostName Property, Options Property, RemotePort Property, Disconnect Method, OnConnect Event

CreateArticle Method

Creates a new article in the current newsgroup.

Syntax

object.**CreateArticle**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CreateArticle** method sends the POST command to the news server. Not all servers permit clients to post articles. The client application is responsible for generating the message headers as well as the body of the message. News articles conform to the same general characteristics of an email message.

The **CloseArticle** method must be called once the contents of the article has been written to the server.

See Also

GroupName Property, CloseArticle Method, OpenArticle Method, Write Method

Disconnect Method

Terminate the connection with a server.

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

IsConnected Property, Connect Method, OnDisconnect Event

GetArticle Method

Retrieve an article from the server.

Syntax

object.**GetArticle**(*Article*, *Message*, [*Options*])

Parameters

Article

Number of the article to retrieve from the server, or a string that specifies the message ID of the article to retrieve. If an article number is specified, this value must be greater than zero. The first available article in the newsgroup can be determined by checking the value of the **FirstArticle** property. The last available article in the newsgroup is returned by the **LastArticle** property.

Message

A string or byte array which will contain the data transferred from the server when the method returns.

Options

An optional integer value which specifies one or more options. This argument is reserved for future use and should be omitted.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **GetArticle** method is used to retrieve an article from the server and copy it into a local buffer. This method will cause the current thread to block until the article transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

FirstArticle Property, LastArticle Property, CreateArticle Method, GetHeaders Method, OpenArticle Method, StoreArticle Method, OnProgress Event

GetFirstArticle Method

Return information about the first article available in the current newsgroup.

Syntax

object.**GetFirstArticle**(*Article*, [*Subject*], [*Author*], [*Posted*], [*MessageId*], [*References*], [*ByteCount*], [*LineCount*])

Parameters

Article

An integer value which specifies the article ID. This is the number that should be used to access the article on the server. This parameter is required and must be passed by reference.

Subject

An optional string which specifies the subject of the article. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

Author

An optional string which specifies the author of the article. This is typically the name and email address of the user who posted the article. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

Posted

An optional string which specifies the date that the article was posted. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

MessageId

An optional string which specifies the message ID for the article. Although it is more common to reference an article by number, it is possible to reference an article by its message ID. To select a message by its message ID string, set the **MessageID** property. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

References

An optional string which specifies references to the article. This can be used by an application to create a list of cross references to the article so that related threads can be provided to the user. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

ByteCount

An optional integer value which specifies the size of the message in bytes. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

LineCount

An optional integer value which specifies the number of lines of text in the message. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

Return Value

A value of true is returned if the operation was successful, otherwise a return value of false indicates that there are no articles available in the currently selected newsgroup.

Remarks

The **GetFirstArticle** method returns information about the first article in the currently selected newsgroup. This method is used in conjunction with the **GetNextArticle** method to enumerate all of

the articles in the newsgroup. Typically this is used to provide the user with a list of articles to access.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **GetArticle** method cannot be called while inside a loop calling **GetNextArticle**. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **GetArticle** for each article number to retrieve the article text.

A program should use either the **ListArticles** method or the **GetFirstArticle** and **GetNextArticle** methods, but never in combination with one another.

Example

```
Dim nArticleId As Long
Dim strSubject As String
Dim strAuthor As String
Dim datePosted As Date
Dim strMessageId As String
Dim strReferences As String
Dim nByteCount As Long
Dim nLineCount As Long
Dim bResult As Boolean;

' List each article in the current newsgroup, adding the subject to
' a ListBox control
bResult = NntpClient1.GetFirstArticle(nArticleId, strSubject, strAuthor, _
    datePosted, strMessageId, strReferences, _
    nByteCount, nLineCount);

Do While bResult
    List1.AddItem strSubject
    List1.ItemData(List1.NewIndex) = nArticleId

    bResult = NntpClient1.GetNextArticle(nArticleId, strSubject, strAuthor, _
        datePosted, strMessageId, strReferences, _
        nByteCount, nLineCount);
End Do
```

See Also

Article Property, MessageID Property, GetNextArticle Method, ListArticles Method,

GetFirstGroup Method

Return information about the first available newsgroup.

Syntax

object.**GetFirstGroup**(*GroupName*, [*FirstArticle*], [*LastArticle*], [*Access*])

Parameters

GroupName

An optional string which specifies the name of the newsgroup. This parameter is required and must be passed by reference.

FirstArticle

An optional integer value which specifies the number for the first available article in the newsgroup. This value corresponds to the **FirstArticle** property of a selected newsgroup. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

LastArticle

An optional integer value which specifies the number for the last available article in the newsgroup. This value corresponds to the **LastArticle** property of a selected newsgroup. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

Access

AAn optional integer value which specifies the access rights for the newsgroup. This parameter must be passed by reference, or it may be omitted if the program does not require this information. It may be one of the following values:

Value	Description
nntpGroupReadOnly	The group is read-only and cannot be modified. Attempts to post articles to the newsgroup will result in an error.
nntpGroupReadWrite	Articles can be posted to the newsgroup. Even though a newsgroup is read-write, it may require that the client authenticate before being given permission to post articles to the server.
nntpGroupModerated	The newsgroup is moderated and articles can only be posted by the group moderator. To request that an article be posted to the newsgroup, you must email the message to the moderator.

Return Value

A value of true is returned if the operation was successful, otherwise a return value of false indicates that there are no newsgroups available on the server. Note that if no newsgroups are returned by the server, it may indicate that it requires the client to authenticate itself prior to requesting a list of groups or articles.

Remarks

The **GetFirstGroup** method returns information about the first newsgroup on the server. This method is used in conjunction with the **GetNextGroup** method to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select. If the **LastUpdate** property is set, then only newsgroups that have been created since that date will be returned.

While the the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve

articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

A program should use either the **ListGroups** method or the **GetFirstGroup** and **GetNextGroup** methods, but never in combination with one another.

Example

```
Dim strGroupName As String
Dim bResult As Boolean;

' List each newsgroup available on the server, adding the group name
' to a ListBox control
bResult = NntpClient1.GetFirstGroup(strGroupName)

Do While bResult
    List1.AddItem strGroupName
    bResult = NntpClient1.GetNextGroup(strGroupName)
End Do
```

See Also

GroupName Property, GroupTitle Property, LastUpdate Property, GetNextGroup Method, ListGroups Method

GetHeaders Method

Retrieves the headers for the specified article from the server.

Syntax

object.**GetHeaders**(*Article*, *Headers*)

Parameters

Article

Number of article to retrieve from the server, or a string that specifies the message ID of the article header to retrieve. If an article number is specified, this value must be greater than zero. The first available article in the newsgroup can be determined by checking the value of the **FirstArticle** property. The last available article in the newsgroup is returned by the **LastArticle** property.

Headers

A string or byte array which will contain the data transferred from the server when the method returns. This parameter must be passed by reference.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **GetHeaders** method is used to retrieve an article header block from the server and copy it into a local buffer. This method will cause the current thread to block until the article transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

CreateArticle Method, GetArticle Method, OpenArticle Method, OnProgress Event

GetNextArticle Method

Return information about the next article available in the current newsgroup.

Syntax

object.**GetNextArticle**(*Article*, [*Subject*], [*Author*], [*Posted*], [*MessageId*], [*References*], [*ByteCount*], [*LineCount*])

Parameters

Article

An integer value which specifies the article ID. This is the number that should be used to access the article on the server. This parameter is required and must be passed by reference.

Subject

An optional string which specifies the subject of the article. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

Author

An optional string which specifies the author of the article. This is typically the name and email address of the user who posted the article. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

Posted

An optional string which specifies the date that the article was posted. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

MessageId

An optional string which specifies the message ID for the article. Although it is more common to reference an article by number, it is possible to reference an article by its message ID. To select a message by its message ID string, set the **MessageID** property. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

References

An optional string which specifies references to the article. This can be used by an application to create a list of cross references to the article so that related threads can be provided to the user. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

ByteCount

An optional integer value which specifies the size of the message in bytes. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

LineCount

An optional integer value which specifies the number of lines of text in the message. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

Return Value

A value of true is returned if the operation was successful, otherwise a return value of false indicates that there are no more articles available in the currently selected newsgroup.

Remarks

The **GetNextArticle** method returns information about the next available article in the currently selected newsgroup. This method is used in conjunction with the **GetFirstArticle** method to

enumerate all of the articles in the newsgroup. Typically this is used to provide the user with a list of articles to access.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **GetArticle** method cannot be called while inside a loop calling **GetNextArticle**. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **GetArticle** for each article number to retrieve the article text.

A program should use either the **ListArticles** method or the **GetFirstArticle** and **GetNextArticle** methods, but never in combination with one another.

Example

```
Dim nArticleId As Long
Dim strSubject As String
Dim strAuthor As String
Dim datePosted As Date
Dim strMessageId As String
Dim strReferences As String
Dim nByteCount As Long
Dim nLineCount As Long
Dim bResult As Boolean;

' List each article in the current newsgroup, adding the subject to
' a ListBox control
bResult = NntpClient1.GetFirstArticle(nArticleId, strSubject, strAuthor, _
    datePosted, strMessageId, strReferences, _
    nByteCount, nLineCount);

Do While bResult
    List1.AddItem strSubject
    List1.ItemData(List1.NewIndex) = nArticleId

    bResult = NntpClient1.GetNextArticle(nArticleId, strSubject, strAuthor, _
        datePosted, strMessageId, strReferences, _
        nByteCount, nLineCount);
End Do
```

See Also

Article Property, MessageID Property, GetFirstArticle Method, ListArticles Method,

GetNextGroup Method

Return information about the next available newsgroup.

Syntax

object.**GetNextGroup**(*GroupName*, [*FirstArticle*], [*LastArticle*], [*Access*])

Parameters

GroupName

An optional string which specifies the name of the newsgroup. This parameter is required and must be passed by reference.

FirstArticle

An optional integer value which specifies the number for the first available article in the newsgroup. This value corresponds to the **FirstArticle** property of a selected newsgroup. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

LastArticle

An optional integer value which specifies the number for the last available article in the newsgroup. This value corresponds to the **LastArticle** property of a selected newsgroup. This parameter must be passed by reference, or it may be omitted if the program does not require this information.

Access

AAn optional integer value which specifies the access rights for the newsgroup. This parameter must be passed by reference, or it may be omitted if the program does not require this information. It may be one of the following values:

Value	Description
nntpGroupReadOnly	The group is read-only and cannot be modified. Attempts to post articles to the newsgroup will result in an error.
nntpGroupReadWrite	Articles can be posted to the newsgroup. Even though a newsgroup is read-write, it may require that the client authenticate before being given permission to post articles to the server.
nntpGroupModerated	The newsgroup is moderated and articles can only be posted by the group moderator. To request that an article be posted to the newsgroup, you must email the message to the moderator.

Return Value

A value of true is returned if the operation was successful, otherwise a return value of false indicates that there are no more newsgroups available on the server. Note that if no newsgroups are returned by the server, it may indicate that it requires the client to authenticate itself prior to requesting a list of groups or articles.

Remarks

The **GetNextGroup** method returns information about the next available newsgroup on the server. This method is used in conjunction with the **GetFirstGroup** method to enumerate all of the available newsgroups. Typically this is used to provide the user with a list of newsgroups to select. If the **LastUpdate** property is set, then only newsgroups that have been created since that date will be returned.

While the the newsgroups are being listed, the client cannot select a newsgroup or retrieve the

contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

A program should use either the **ListGroups** method or the **GetFirstGroup** and **GetNextGroup** methods, but never in combination with one another.

Example

```
Dim strGroupName As String
Dim bResult As Boolean;

' List each newsgroup available on the server, adding the group name
' to a ListBox control
bResult = NntpClient1.GetFirstGroup(strGroupName)

Do While bResult
    List1.AddItem strGroupName
    bResult = NntpClient1.GetNextGroup(strGroupName)
End Do
```

See Also

GroupName Property, GroupTitle Property, LastUpdate Property, GetFirstGroup Method, ListGroups 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 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 nntpClient = CreateObject("SocketTools.NntpClient.11")

nError = nntpClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize Method

ListArticles Method

Return a list of articles in the current newsgroup.

Syntax

object.**ListArticles**([*FirstArticle*], [*LastArticle*])

Parameters

FirstArticle

An optional integer argument which specifies the first article to list. If this argument is omitted, the list will start with the first available article in the newsgroup.

LastArticle

An optional integer argument which specifies the last article to list. If this argument is omitted, the list will end with the last available article in the newsgroup.

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 **ListArticles** method requests that the server return a list of news articles in the specified range for the current newsgroup. The **OnNewsArticle** event will be fired for each article returned by the server. When the last news article has been listed, the **OnLastArticle** event will be fired.

While the articles in the newsgroup are being listed, the client cannot retrieve the contents of a specific article. For example, the **GetArticle** method cannot be called while inside the **OnNewsArticle** event. The client should store those articles which it wants to retrieve in an array, and then once all of the articles have been listed, it can begin calling **GetArticle** for each article number to retrieve the article text.

A program should use either the **ListArticles** method or the **GetFirstArticle** and **GetNextArticle** methods, but never in combination with one another.

See Also

FirstArticle Property, LastArticle Property, GetFirstArticle Method, GetNextArticle Method, SelectGroup Method, OnLastArticle Event, OnNewsArticle Event

ListGroups Method

Return a list of newsgroups available on the server.

Syntax

object.**ListGroups**([*LastUpdate*])

Parameters

LastUpdate

An optional string argument which specifies the date and time that the list of newsgroups were last retrieved from the server. If this argument is omitted, then the value of the **LastUpdate** property will be used. If the value is an empty string, all available newsgroups will be listed by the server.

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 **ListGroups** method requests that the server return a list of available news groups. The **OnNewsGroup** event will be fired for each group returned by the server. When the last news group has been listed by the control, the **OnLastGroup** event will fire.

While the the newsgroups are being listed, the client cannot select a newsgroup or retrieve the contents of a specific article. The client should store those newsgroups which it wants to retrieve articles from, and then once all of the newsgroups have been listed, it can then select each newsgroup and retrieve the available articles from that group.

A program should use either the **ListGroups** method or the **GetFirstGroup** and **GetNextGroup** methods, but never in combination with one another.

See Also

GroupName Property, LastUpdate Property, GetFirstGroup Property, GetNextGroup Property, OnLastGroup Event, OnNewsGroup Event

OpenArticle Method

Opens the specified article in the currently selected newsgroup.

Syntax

object.**OpenArticle**(*Article*, [*Options*])

Parameters

Article

Number of the article to retrieve from the server. This value must be greater than zero. The first available article in the newsgroup can be determined by checking the value of the **FirstArticle** property. The last available article in the newsgroup is returned by the **LastArticle** property.

Options

An optional integer value which specifies one or more options. This argument is reserved for future use and should be omitted.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **OpenArticle** method opens the specified article in the currently selected newsgroup. The article data can be read using the **Read** method, and once all of the data has been returned, the **CloseArticle** method should be used to close the article on the server.

See Also

FirstArticle Property, LastArticle Property, CloseArticle Method, CreateArticle Method, Read Method

PostArticle Method

Post a new article to the current newsgroup.

Syntax

object.**PostArticle**(*Message*, [*Options*])

Parameters

Message

A string or byte array which will contain the article to be posted to the server.

Options

An optional integer value which specifies one or more options. This argument is reserved for future use and should be omitted.

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 **PostArticle** method is used to submit the contents of the specified buffer to the server as a new article in the current newsgroup. This method will cause the current thread to block until the article transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

Not all newsgroups permit new articles to be posted, and some newsgroups may require that you email the article to a moderator for approval instead of posting directly to the group. It may be required that the client authenticate itself using the **Authenticate** method prior to posting the article.

A news article is similar to an email message in that it contains one or more header fields, followed by an empty line, followed by the body of the article. Each line of text should be terminated by a carriage return/linefeed sequence of characters. The Mail Message control can be used to compose a message if needed. Note that the article header must contain a header field named "Newsgroups" with a value that specifies the newsgroup or newsgroups the article is being posted to. If this header field is missing, the news server will reject the article.

See Also

Authenticate Method, GetArticle Method, GetHeaders Method, OnProgress Event

Read Method

Return data read from the server.

Syntax

object.**Read**(*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

The number of bytes actually read from the server is returned by this method. If an error occurs, NNTP_ERROR is returned.

Remarks

The **Read** method returns data that has been read from the server, 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.

See Also

IsConnected Property, IsReadable Property, Write Method, OnRead Event, OnWrite 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

SelectGroup Method

Selects the specified newsgroup as the current newsgroup.

Syntax

object.**SelectGroup**(*GroupName*)

Parameters

GroupName

A string which specifies the name of the newsgroup to select.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **SelectGroup** method selects a newsgroup and updates the control with information about the group. The list of available newsgroups can be enumerated using the **ListGroups** method.

See Also

FirstArticle Property, GroupName Property, LastArticle Property, ListGroups Method

StoreArticle Method

Retrieve an article from the current newsgroup and store it in a local file.

Syntax

object.**StoreArticle**(*Article*, *FileName*)

Parameters

Article

Number of the article to retrieve from the server. This value must be greater than zero. The first available article in the newsgroup can be determined by checking the value of the **FirstArticle** property. The last available article in the newsgroup is returned by the **LastArticle** property.

FileName

Pointer to a string which specifies the file that the article will be stored in. If an empty string is passed as an argument, the article is copied to the system clipboard.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **StoreArticle** method retrieves an article from the server and stores it in a file on the local system. The contents of the article is stored as a text file, using the specified file name. This method will cause the current thread to block until the message transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

GetArticle Method, OpenArticle 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. 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 server.

Syntax

object.**Write**(*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 because 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 you wish to send the data without conversion, use a **Byte** array as the buffer instead of a **String** variable.

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 server, or NNTP_ERROR if an error was encountered.

Remarks

The **Write** method sends the data in ***buffer*** to the server. 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 blocking, the application will wait until the data can be sent. If the control is non-blocking and the write fails because it could not send all of the data to the server, the **OnWrite** event will be fired when the server can accept data again.

See Also

IsConnected Property, IsWritable Property, Timeout Property, Read Method, OnWrite Event

Network News Transfer Protocol Control Events

Event	Description
OnCancel	This event is generated when a blocking operation is canceled
OnCommand	This event is generated when the server processes a command issued by the client
OnConnect	This event is generated when a connection is established
OnDisconnect	This event is generated when a connection is terminated
OnError	This event is generated when a control error occurs
OnLastArticle	This event is generated after the last news article has been returned by the server
OnLastGroup	This event is generated after the last newsgroup has been returned by the server
OnNewsArticle	This event is generated when the server returns information about an article
OnNewsGroup	This event is generated when the server returns information about a newsgroup
OnProgress	This event is generated during data transfer
OnRead	This event is generated when data is available to be read
OnWrite	This event is generated when data can be written to the server

OnCancel Event

The **OnCancel** event is generated when a blocking operation is canceled.

Syntax

Sub *object*_**OnCancel** ([*Index* **As Integer**])

Remarks

This event is generated 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, OnTimeout Event

OnCommand Event

The **OnCommand** event is generated when the client sends a command to the server and receives a reply indicating the results of that command.

Syntax

Sub *object*_**OnCommand**([*Index* **As Integer**], **ByVal** *ResultCode* **As Variant**, **ByVal** *ResultString* **As Variant**)

Remarks

The **OnCommand** event is generated when the client receives a reply from the server after some action has been taken. The ***ResultCode*** argument contains the numeric result code returned by the server. The result codes returned from the server fall into one of the following categories:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

The ***ResultString*** argument contains the descriptive string returned by the server which describes the result. The string contents may vary depending on the type of server.

See Also

ResultCode Property, ResultString Property, Command Method

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 server as a result of a **Connect** method call. This event is only triggered when the **Blocking** property is set to False.

See Also

Blocking Property, Connect Method, OnDisconnect Event, OnWrite 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. This event is only triggered when the **Blocking** property is set to False.

When the **OnDisconnect** event fires, it is possible that there may still be buffered data available to read from the server. Before disconnecting from the server, the application should attempt to read any remaining data until the **Read** method returns a value of zero, or returns an error indicating that the operation would block.

See Also

Blocking Property, IsConnected Property, IsReadable Property, Connect Method, Disconnect Method, Read Method, 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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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 component 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

OnLastArticle Event

The **OnLastArticle** event is generated after the last news article has been returned by the server.

Syntax

Sub *object*_**OnLastArticle**([*Index* **As Integer**])

Remarks

The **OnLastArticle** event is fired after the server has returned all of the articles in the current newsgroup as a result of the application calling the **ListArticles** method.

See Also

ListArticles Method, OnNewsArticle Event

OnLastGroup Event

The **OnLastGroup** event is generated after the last newsgroup has been returned by the server.

Syntax

Sub *object*_**OnLastGroup**([*Index* **As Integer**])

Remarks

The **OnLastGroup** event is fired after the server has returned all of the available newsgroups as a result of the application calling the **ListGroups** method.

See Also

ListGroups Method, OnNewsGroup Event

OnNewsArticle Event

The **OnNewsArticle** event is generated when the server returns information about an article.

Syntax

Sub *object*_**OnNewsArticle**([*Index* **As Integer**,] **ByVal** *Article* **As Variant**, **ByVal** *Subject* **As Variant**, **ByVal** *Author* **As Variant**, **ByVal** *Posted* **As Variant**, **ByVal** *MessageId* **As Variant**, **ByVal** *References* **As Variant**, **ByVal** *ByteCount* **As Variant**, **ByVal** *LineCount* **As Variant**)

Remarks

The **OnNewsArticle** event is generated when the server returns information about an article in the current newsgroup. Calling the **ListArticles** method causes this event to be generated for each article in the newsgroup. The following arguments are passed to the event handler:

Article

An integer value which specifies the article ID. This is the number that should be used to access the article on the server.

Subject

A string value which specifies the subject of the article.

Author

A string value which specifies the author of the article. This is typically the name and email address of the user who posted the article.

Posted

A string value which specifies the date that the article was posted.

MessageId

A string value which specifies the message ID for the article. Although it is more common to reference an article by number, it is possible to reference an article by its message ID. To select a message by its message ID string, set the **MessageID** property.

References

A string which specifies references to the article. This can be used by an application to create a list of cross references to the article so that related threads can be provided to the user.

ByteCount

An integer value which specifies the size of the message in bytes.

LineCount

An integer value which specifies the number of lines of text in the message.

See Also

Article Property, MessageID Property, ListArticles Method, OnLastArticle Event

OnNewsGroup Event

The **OnNewsGroup** event is generated when the server returns information about a newsgroup.

Syntax

Sub *object*_**OnNewsGroup**([*Index* **As Integer**,] **ByVal** *GroupName* **As Variant**, **ByVal** *FirstArticle* **As Variant**, **ByVal** *LastArticle* **As Variant**, **ByVal** *Access* **As Variant**)

Remarks

The **OnNewsGroup** event is generated when the server returns information about a newsgroup. Calling the **ListGroups** method causes this event to be generated for each newsgroup on the server. The following arguments are passed to the event handler:

GroupName

An string value which specifies the name of the newsgroup.

FirstArticle

An integer value which specifies the number for the first available article in the newsgroup. This value corresponds to the **FirstArticle** property of a selected newsgroup.

LastArticle

An integer value which specifies the number for the last available article in the newsgroup. This value corresponds to the **LastArticle** property of a selected newsgroup.

Access

An integer value which specifies the access rights for the newsgroup. It may be one of the following values:

Value	Description
nntpGroupReadOnly	The group is read-only and cannot be modified. Attempts to post articles to the newsgroup will result in an error.
nntpGroupReadWrite	Articles can be posted to the newsgroup. Even though a newsgroup is read-write, it may require that the client authenticate before being given permission to post articles to the server.
nntpGroupModerated	The newsgroup is moderated and articles can only be posted by the group moderator. To request that an article be posted to the newsgroup, you must email the message to the moderator.

See Also

GroupName Property, GroupTitle Property, ListGroups Method, OnLastGroup Event

OnProgress Event

The **OnProgress** event is generated during data transfer.

Syntax

Sub *object*_**OnProgress** ([*Index* **As Integer**], **ByVal** *Article* **As Variant**, **ByVal** *ArticleSize* **As Variant**, **ByVal** *ArticleCopied* **As Variant**, **ByVal** *Percent* **As Variant**)

Remarks

The **OnProgress** event is generated during the transfer of data between the client and server, indicating the amount of data exchanged. For transfers of 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:

Article

An integer value which specifies the number of the article being retrieved. If an article is being posted, this argument will have a value of zero.

ArticleSize

The size of the article being transferred in bytes.

ArticleCopied

The number of bytes that have been transferred between the client and server.

Percent

The percentage of data that's been transferred, expressed as an integer value between 0 and 100, inclusive. If the size of the file on the server cannot be determined, this value will always be 100.

Note that this event is only generated when a news article is transferred using the **GetArticle** or **PostArticle** methods. If the client is reading or writing the file data directly to the server using the **Read** or **Write** methods then the application is responsible for calculating the completion percentage and updating any user interface controls.

See Also

GetArticle Method, PostArticle 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 triggered when the **Blocking** property is set to False.

See Also

IsReadable Property, 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. This event is only triggered when the **Blocking** property is set to True.

See Also

Timeout Property, OnCancel Event

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 triggered when the **Blocking** property is set to False.

See Also

IsWritable Property, Read Method, Write Method, OnConnect Event, OnRead Event

Post Office Protocol Control

List and retrieve email messages from a mail server.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	PopClientCtl.PopClient
File Name	CSPOPX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.PopClient.11
ClassID	74FE8C6C-4C7F-40DC-9BE7-23F049EF3948
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 1939

Overview

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

This control supports secure connections using the TLS protocol. Both implicit and explicit TLS connections can be established, enabling the control to work with a wide variety of servers.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows

operating system.

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.

Post Office Protocol Control Properties

Property	Description
AuthType	Gets and sets the method used to authenticate the user
AutoResolve	Determines if host names and IP addresses are automatically resolved
BearerToken	Gets and sets the OAuth 2.0 bearer token used for authentication
Blocking	Gets and sets the blocking state of the control
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 client certificate
CertificatePassword	Gets and sets the password associated with the client certificate
CertificateStatus	Return the status of the server certificate
CertificateStore	Gets and sets the name of the client 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
HashStrength	Return the length of the message digest that was selected
HeaderField	Gets and sets the current header field name
HeaderValue	Return the value of the current header field
HostAddress	Gets and sets the IP address of the mail server
HostName	Gets and sets the host name of the mail server
IsBlocked	Return if the control is blocked performing an operation
IsConnected	Determine if the control is connected to a server
IsInitialized	Determine if the control has been initialized
IsReadable	Return if data can be read from the server without blocking
IsWritable	Return if data can be sent to the server without blocking
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error to occur
LastMessage	Return the number of the last message available on the server
MailboxSize	Return the size of the current mailbox
Message	Gets and sets the current message number
MessageCount	Return the number of messages available in the current mailbox
MessageFrom	Return the address of the user who sent the message
MessageUID	Return the unique ID for the current message on the mail server
MessageSize	Return the size of the current message in bytes

Options	Gets and sets the options that are used in establishing a connection
Password	Gets and sets the password for the current user
RemotePort	Gets and sets the port number for a remote connection
ResultCode	Return the result code of the previous action
ResultString	Return a string describing the results of the previous action
Secure	Set or return if a connection to the server is secure
SecureCipher	Return the encryption algorithm used to establish the secure connection with the server
SecureHash	Return the message digest selected when establishing the secure connection with the server
SecureKeyExchange	Return the key exchange algorithm used to establish the secure connection with the server
SecureProtocol	Gets and sets the security protocol used to establish the secure connection with the server
ThrowError	Enable or disable error handling by the container of the control
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
UserName	Gets and sets the current user name
Version	Return the current version of the object

AuthType Property

Gets and sets the method used to authenticate the user.

Syntax

object.**AuthType** [= *type*]

Remarks

The **AuthType** property specifies the type of authentication that should be used when the client connects to the mail server. The following authentication methods are supported:

Value	Description
popAuthPass	The username and password is sent to the server using the USER and PASS commands. This authentication method is supported by most servers and is the default authentication type. The credentials are not encrypted and this method should only be used over secure connections.
popAuthApop	The APOP authentication method which uses an MD5 digest of the password. This method has been deprecated is not supported by all servers. It should only be used if required by legacy mail servers which do not support the SASL authentication methods.
popAuthLogin	This authentication type will use the AUTH LOGIN command to authenticate the client session. This encodes the username and password in a specific format, but the credentials are not encrypted. It should be used over a secure connection. The server must support the Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4422.
popAuthPlain	This authentication type will use the AUTH PLAIN command to authenticate the client session. This encodes the username and password in a specific format, but the credentials are not encrypted. It should be used over a secure connection. The server must support the PLAIN Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4616.
popAuthXOAuth2	This authentication type will use the AUTH XOAUTH2 command to authenticate the client session. This authentication method does not require the user password, instead the **BearerToken** property must specify the bearer token issued by the service provider.
popAuthBearer	This authentication type will use the AUTH OAUTHBEARER command to authenticate the client session as defined in RFC 7628. This authentication method does not require the user password, instead the **BearerToken** property must specify the bearer token issued by the service provider.

Data Type

Integer (Int32)

Remarks

The **popAuthLogin** and **popAuthPlain** authentication methods require the mail server support the

Simple Authentication and Security Layer (SASL) AUTH command as defined in RFC 5034. Most modern mail servers do support one or both of these methods, and they are generally preferred over the **popAuthPass** method when possible. However, for backwards compatibility with legacy servers, the API will default to using **popAuthPass** for client authentication.

The **popAuthXOAuth2** and **popAuthBearer** authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Changing the value of the **BearerToken** property will automatically set the current authentication method to use OAuth 2.0.

See Also

BearerToken Property, Password Property, UserName Property, Connect Method

AutoResolve Property

Determines if host names and IP 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 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 **AutoResolve** to False.

Data Type

Boolean

See Also

HostAddress Property, HostName Property

BearerToken Property

Gets and sets the OAuth 2.0 bearer token for the current user.

Syntax

object.**BearerToken** [= *token*]

Remarks

The **BearerToken** property specifies the OAuth 2.0 bearer token used to authenticate the user. This property is used as the default value for the **Connect** method if the token is not provided as an parameter.

Assigning a value to this property will change the current authentication method to use OAuth 2.0 if necessary.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an bearer token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

If the current authentication method does not use OAuth 2.0, this property will return an empty string and you should check the value of the **Password** property to obtain the current user's password. Refer to the **AuthType** property for more information on the available authentication methods.

Data Type

String

See Also

AuthType Property, Password Property, UserName Property, Connect 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 read data when none has been written, an error is generated. Events such as **OnConnect**, **OnDisconnect**, OnRead and **OnWrite** are only fired if the connection is non-blocking.

Data Type

Boolean

See Also

IsBlocked Property, IsReadable Property, IsWritable Property

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 = PopClient1.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 client certificate.

Syntax

object.**CertificateName** [= *name*]

Remarks

This property sets the common name or friendly name of the certificate that should be used to establish the connection with the server. 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. If a certificate name is specified, the certificate must have a private key associated with it, otherwise the connection attempt 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 client 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
stCertificateNone	No certificate information is available. A secure connection was not established with the server.
stCertificateValid	The certificate is valid.
stCertificateNoMatch	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.
stCertificateExpired	The certificate has expired and is no longer valid. The client can examine the **CertificateExpires** property to determine when the certificate expired.
stCertificateRevoked	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.
stCertificateUntrusted	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.
stCertificateInvalid	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:

```
'
' Initialize the control properties
```

```
'

PopClient1.HostName = strHostName
PopClient1.Secure = True

nError = PopClient1.Connect()
If nError > 0 Then
     MsgBox "Unable to connect to server " & strHostName, vbExclamation
     Exit Sub
End If

If PopClient1.CertificateStatus <> stCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          PopClient1.Disconnect
          Exit Sub
     End If
End If

PopClient1.Disconnect
```

See Also

CertificateExpires Property, CertificateIssued Property, CertificateIssuer Property, CertificateSubject Property, Secure Property

CertificateStore Property

Gets and sets the name of the client certificate store or file.

Syntax

object.**CertificateStore** [= *store*]

Remarks

This property sets the name of the certificate store that contains the client certificate that should be used when establishing a secure connection with 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 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

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 = PopClient1.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 client certificate.

Syntax

object.**CertificateUser** [= *username*]

Remarks

This property sets the name of the user that owns the client certificate that will be used to establish a secure connection with the server. 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

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

HeaderField Property

Gets and sets the current header field name.

Syntax

object.**HeaderField** [= *header*]

Remarks

The **HeaderField** property returns the name of the current header field. Setting this property causes the control to determine if that header field exists, and if it does, to update the **HeaderValue** property with its value. This property can be used to obtain the value of any header in the current message.

Note that the server must support the XTND XLST command in order to retrieve the header value. If the command is not supported, the **HeaderValue** property will return an empty string.

Data Type

String

See Also

HeaderValue Property, Message Property, MessageUID Property

HeaderValue Property

Return the value of the current header field.

Syntax

object.**HeaderValue**

Remarks

The **HeaderValue** property returns the value of the header field specified by the **HeaderField** property. This property can be used to obtain the value of any header in the current message.

Note that the server must support the XTND XLST command in order to retrieve the header value. If the command is not supported, the **HeaderValue** property will return an empty string.

Data Type

String

See Also

HeaderField Property, Message Property, MessageUID Property

HostAddress Property

Gets and sets the IP address of the server.

Syntax

object.**HostAddress** [= *ipaddress*]

Remarks

The **HostAddress** property can be used to set the IP address for a server 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, HostName Property

HostName Property

Gets and sets the name of the server.

Syntax

object.**HostName** [= *hostname*]

Remarks

The **HostName** property should be set to the name of the server 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

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.

Note that this property will return True if there is ***any*** blocking operation being performed by the application, regardless if the specified control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

Blocking Property, LastError Property

IsConnected Property

Determine if the control is connected to a server.

Syntax

object.**IsConnected**

Remarks

The **IsConnected** read-only property is set to a value of true if the control is connected with a server, otherwise the property has a value of false.

Data Type

Boolean

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

IsReadable Property

Return if data can be read from the server without blocking.

Syntax

object.**IsReadable**

Remarks

The **IsReadable** property returns True if data can be read from the server without blocking. For non-blocking connections, this property can be checked before the application attempts to read the data, preventing an error.

Data Type

Boolean

See Also

IsConnected Property, Read Method, OnRead Event

IsWritable Property

Return if data can be sent to the server without blocking.

Syntax

object.**IsWritable**

Remarks

The **IsWritable** property returns True if data can be written without blocking. For non-blocking connections, this property can be checked before the application attempts to send data to the server, 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 send data without an error. The next operation may result in an **stErrorOperationWouldBlock** or **stErrorOperationInProgress** error. The application must treat these errors as recoverable, and should be prepared to retry operations that result in them.

Data Type

Boolean

See Also

IsReadable Property, Write Method, OnWrite Event

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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

LastMessage Property

Return the number of the last message available on the server.

Syntax

object.**LastMessage**

Remarks

The **LastMessage** property returns the last message available on the server. Note that unlike the **MessageCount** property, this value remains constant even when a message is deleted.

Data Type

Integer (Int32)

See Also

Message Property, MessageCount Property, MessageSize Property

MailboxSize Property

Return the size of the current mailbox.

Syntax

object.**MailboxSize**

Remarks

The **MailboxSize** property returns the combined size in bytes of all of the available messages in the current mailbox. Note that as messages are deleted from the mailbox, this property value will decrease.

Data Type

Integer (Int32)

See Also

Message Property, MessageCount Property, MessageSize Property

Message Property

Gets and sets the current message number.

Syntax

object.**Message** [= *value*]

Remarks

The **Message** property sets or returns the message number for the currently selected mailbox. Message numbers range from 1 through the number of messages available on the server, as returned by the **MessageCount** property. Setting the **Message** property to an invalid message number will generate an error.

Data Type

Integer (Int32)

See Also

MessageCount Property, MessageSize Property

MessageCount Property

Return the number of messages available in the current mailbox.

Syntax

object.**MessageCount**

Remarks

The **MessageCount** property returns the number of messages available to be retrieved from the currently selected mailbox.

Data Type

Integer (Int32)

See Also

Message Property, MessageSize Property, MessageUID Property, GetMessage Method

MessageFrom Property

Return the address of the user who sent the message.

Syntax

object.**MessageFrom**

Remarks

The **MessageFrom** property returns the address of the user who sent the current message. This property uses either the XSENDER or the XTND XLST command in order to determine who the sender is. The XSENDER command returns an authenticated address, as used with the Netscape SMTP authentication method. If this command is not supported, or the sender's address was not authenticated, then the XTND XLST command is used to return the value of the "From" header field in the message. If this command is not supported by the server, the method will attempt to retrieve the entire message header and return the value for the specified header field. This enables an application to use this property even if the server does not support command extensions.

Data Type

String

See Also

Message Property, MessageCount Property, MessageSize Property

MessageSize Property

Return the size of the current message in bytes.

Syntax

object.**MessageSize**

Remarks

The **MessageSize** property returns the size of the current message in bytes. The size includes the header and body portion of the message.

Data Type

Integer (Int32)

See Also

Message Property, MessageCount Property, MessageUID Property

MessageUID Property

Return the unique ID for the current message on the mail server.

Syntax

object.**MessageUID**

Remarks

The **MessageID** property returns a string which uniquely identifies the message on the server. The identifier is assigned by the mail server, and retains the same value across multiple client sessions. This value is typically used when the client wants to leave a message on the mail server, but does not wish to retrieve the message contents multiple times. For example, the client can store the unique ID for each message that it retrieves, but does not delete from the server. The next time that it connects to the mail server, it compares the unique ID of a message against the stored values. If there is a match, the client knows that the message has already been retrieved, and does not need to do so again.

This property requires that the server support the optional UIDL command. If the command is not supported, this property will always return an empty string. Note that the unique ID for the message is not the same as the Message-ID header field in the message itself.

Data Type

String

See Also

Message Property, MessageCount Property, MessageSize Property

Options Property

Gets and sets the options that are used in establishing a connection.

Syntax

object.**Options** [= *value*]

Remarks

The **Options** property is an integer value which specifies one or more options. The value specified for this property will be used as the default options when connecting to the server. The property value is created by using a bitwise operator with one or more of the following values:

Value	Description
popOptionNone	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used.
popOptionLineBreak	Message data that is received from the server is read as individual lines of text terminated by a carriage return and linefeed control sequence. This option can be useful for applications that need to use the lower level network I/O functions and must process the message text on a line-by-line basis. This option is not recommended for most applications because it can have a negative impact on performance when retrieving large messages from the server.
&H400	popOptionTunnel	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
&H800	popOptionTrustedSite	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 the TLS protocol.
&H1000	popOptionSecureExplicit	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. This option initiates the secure session using the STLS command.
&H2000	popOptionSecureImplicit	This option specifies the client should attempt to establish a secure connection with the server. It should only be used when the server expects an implicit TLS connection or does not implement RFC 2595 where the STLS command is used to negotiate a secure connection with the server.
&H8000	popOptionSecureFallback	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.
&H40000	popOptionPreferIPv6	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.

Data Type

Integer (Int32)

See Also

Secure Property, Connect Method

Password Property

Gets and sets the password for the current user.

Syntax

object.**Password** [= *password*]

Remarks

The **Password** property specifies the password used to authenticate the user. This property is used as the default value for the **Connect** method if no password is specified as an argument.

Refer to the **AuthType** property for more information on the available authentication methods. If you are using the OAuth 2.0 authentication method, this property should not be set to the user's password. Instead, you should set the **BearerToken** property to the bearer token issued by the mail service provider. Note that these access tokens can be much larger than your typical password and are only valid for a limited period of time.

You can use the **Password** property to specify an OAuth 2.0 bearer token. However, it is recommended that you use the **BearerToken** property instead of assigning it to this property. It will ensure compatibility with future versions of the control and make it clear in your code you are using a bearer token and not a password. If the **AuthType** property specifies one of the OAuth 2.0 authentication methods, this property will return the bearer token.

Data Type

String

See Also

AuthType Property, BearerToken Property, UserName Property, Connect Method

RemotePort Property

Gets and sets the port number for a remote connection.

Syntax

object.**RemotePort** [= *portnumber*]

Remarks

The **RemotePort** property is used to set the port number that the control will use to establish a connection with the server.

Data Type

Integer (Int32)

See Also

HostAddress Property, HostName Property

ResultCode Property

Return the result code of the previous action.

Syntax

object.**ResultCode**

Remarks

The **ResultCode** read-only property returns the result of the last action performed by the client. This property should be checked after the **Command** method is used to execute a command on the server to determine if the operation was successful.

If the **ResultCode** property returns a value of true, that corresponds to an OK response from the server which indicates that the command was successful. If the property returns a value of false, that corresponds to an ERR response from the server which indicates that the command failed. The **ResultString** property typically returns more detailed information as to why the command failed.

Data Type

Boolean

See Also

ResultString Property, Command Method, OnCommand Event

ResultString Property

Return a string describing the results of the previous action.

Syntax

object.**ResultString**

Remarks

The **ResultString** read-only property returns the result string from the last action taken by the client. This string is generated by the server, and typically is used to describe the result code. For example, if an error is indicated by the result code, the result string may describe the condition that caused the error.

Data Type

String

See Also

ResultCode Property, Command Method, OnCommand Event

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 must 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 strongly recommended that any application that sets this property True use error handling to trap an errors that may occur. If the control 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.

Data Type

Boolean

Example

The following example establishes a secure connection to a server:

```
PopClient1.HostName = strHostName
PopClient1.RemotePort = 110
PopClient1.UserName = strUserName
PopClient1.Password = strPassword
PopClient1.Secure = True

nError = PopClient1.Connect()
If nError > 0 Then
    MsgBox "Unable to connect to server " & strHostName, vbExclamation
    Exit Sub
End If

If PopClient1.CertificateStatus <> stCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          PopClient1.Disconnect
          Exit Sub
     End If
End If
```

See Also

CertificateStatus 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
stCipherNone	No cipher has been selected. This is not a secure connection with the server.
stCipherRC2	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.
stCipherRC4	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.
stCipherRC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
stCipherDES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher using 56-bit keys.
stCipherDES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively using a 168-bit key length.
stCipherDESX	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.
stCipherAES	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.
stCipherSkipjack	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
stCipherBlowfish	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
stHashMD5	The MD5 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA1	The SHA-1 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA256	The SHA-256 algorithm has been selected.
stHashSHA384	The SHA-384 algorithm has been selected.
stHashSHA512	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
stKeyExchangeNone	No key exchange algorithm has been selected. This is not a secure connection with the server.
stKeyExchangeRSA	The RSA public key exchange algorithm has been selected.
stKeyExchangeKEA	The KEA public key exchange algorithm has been selected. This is an improved version of the Diffie-Hellman public key algorithm.
stKeyExchangeDH	The Diffie-Hellman public key exchange algorithm has been selected.
stKeyExchangeECDH	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

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
PopClient1.ThrowError = False
nError = PopClient1.Connect(strHostName)

If nError > 0 Then
    MsgBox PopClient1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

PopClient1.ThrowError = True
PopClient1.Connect strHostName

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error.

Note that the **Timeout** property also determines the amount of time the control will spend attempting to connect to a server. If a connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
popTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
popTraceError	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.
popTraceWarning	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.
popTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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

UserName Property

Gets and sets the current user name.

Syntax

object.**UserName** [= *username*]

Remarks

The **UserName** property specifies the user that is logging in to the server, and is required for authentication purposes. This property is used as the default value for the **Connect** method if no password is specified as an argument.

Data Type

String

See Also

AuthType Property, BearerToken Property, Password Property, Connect Method

Version Property

Return the current version of the object.

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

Post Office Protocol Control Methods

Method	Description
Cancel	Cancels the current blocking network operation
ChangePassword	Change the mailbox password for the current user
CloseMessage	Closes the current message
Command	Send a custom command to the server
Connect	Establish a connection with a server
DeleteMessage	Marks a message for deletion from the mailbox
Disconnect	Terminate the connection with a server
GetHeader	Returns the value of a header field from the specified message
GetHeaders	Retrieves the headers for the specified message from the server
GetMessage	Retrieve a message from the server
Initialize	Initialize the control and validate the runtime license key
OpenMessage	Open a message on the server
Read	Return data read from the server
Reset	Reset the internal state of the control
SendMessage	Submits the contents of a specified file to the mail server for delivery
StoreMessage	Retrieve a message from the server and store it in a local file
Uninitialize	Uninitialize the control and release any system resources that were allocated
Write	Write data to the server

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

Disconnect Method, Reset Method, OnCancel Event

ChangePassword Method

Change the mailbox password for the current user.

Syntax

object.**ChangePassword**(*OldPassword*, *NewPassword*)

Parameters

OldPassword

A string which specifies the user's current mailbox password.

NewPassword

A string which specifies the user's new password. An error will be returned if the old password and the new password are the same 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.

Remarks

The **ChangePassword** method changes the password that will be used to authenticate the user. This method requires that the **UserName** property be set, but it is not required that the user be logged into the POP3 server. Once the password has been changed, the **Password** property will be updated with the new password.

Note that in order to change the user's mailbox password, the server must be running the 'poppass' service on port 106, on the same server. Because passwords are transmitted as clear text (unencrypted), this service is not considered secure and may not be available.

See Also

Password Property, UserName Property

CloseMessage Method

Closes the current message.

Syntax

object.**CloseMessage**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CloseMessage** method closes the current message. If there is any remaining data left to be read from the message, it will be read and discarded.

See Also

OpenMessage Method, Read Method

Command Method

Send a custom command to the server.

Syntax

object.**Command**(*Command*, [*Parameters*], [*Options*])

Parameters

Command

A string which specifies the command to send. Valid commands vary based on the Internet protocol and the type of server that the client is connected to. Consult the protocol standard and/or the technical reference documentation for the server to determine what commands may be issued by a client application.

Parameters

An optional string which specifies one or more parameters to be sent along with the command. If more than one parameter is required, most Internet protocols require that they be separated by a single space character. Consult the protocol standard and/or technical reference documentation for the server to determine what parameters should be provided when issuing a specific command. If no parameters are required for the command, this argument may be omitted.

Options

A numeric value which specifies one or more options. Currently this argument is reserved and should either be omitted, or a value of zero should always be used.

Return Value

A value of zero is returned if the command was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure. To determine the result code returned by the server in response to the command, read the value of the **ResultCode** property.

Remarks

The **Command** method sends a command to the server and processes the result code sent back in response to that command. This method can be used to send custom commands to a server to take advantage of features or capabilities that may not be supported internally by the control.

See Also

ResultCode Property, ResultString Property, OnCommand Event

Connect Method

Establish a connection with a server.

Syntax

object.**Connect**([*RemoteHost*], [*RemotePort*], [*UserName*], [*Password*], [*Timeout*], [*Options*])

Parameters

RemoteHost

A string which specifies the host name or IP address of the server. If this argument is not specified, it defaults to the value of the **HostAddress** property if it is defined. Otherwise, it defaults to the value of the **HostName** property.

RemotePort

A number which specifies the port to connect to on the server. If this argument is not specified, it defaults to the value of the **RemotePort** property. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 110. For secure connections, the default port number is 995. If the secure port number is specified, an implicit TLS connection will be established by default.

UserName

A string which specifies the name of the user used to authenticate access to the server. If this argument is not specified, it defaults to the value of the **UserName** property.

Password

A string which specifies the password used to authenticate the user. If you are using the OAuth 2.0 authentication method, this property should specify the bearer token provided by the mail service and not the user password. Refer to the **AuthType** property for more information about the supported authentication methods. If this argument is not specified, it defaults to the value of the **BearerToken** or **Password** property, depending on the authentication method specified.

Timeout

The number of seconds that the client will wait for a response before failing the operation. If this argument is not specified, the value of the **Timeout** property will be used as the default.

Options

A numeric value which specifies one or more options. If this argument is omitted or a value of zero is specified, a default, standard connection will be established. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description
popOptionNone	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used.
popOptionLineBreak	Message data that is received from the server is read as individual lines of text terminated by a carriage return and linefeed control sequence. This option can

	be useful for applications that need to use the lower level network I/O functions and must process the message text on a line-by-line basis. This option is not recommended for most applications because it can have a negative impact on performance when retrieving large messages from the server.	
&H400	popOptionTunnel	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
&H800	popOptionTrustedSite	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 the TLS protocol.
&H1000	popOptionSecureExplicit	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. This option initiates the secure session using the STLS command.
&H2000	popOptionSecureImplicit	This option specifies the client should attempt to establish a secure connection with the server. It should only be used when the server expects an implicit TLS connection or does not implement RFC 2595 where the STLS command is used to negotiate a secure connection with the server.
&H8000	popOptionSecureFallback	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.
&H40000	popOptionPreferIPv6	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.

See Also

AuthType Property, HostAddress Property, HostName Property, Options Property, RemotePort Property, Disconnect Method, OnConnect Event

DeleteMessage Method

Marks a message for deletion from the mailbox.

Syntax

object.**DeleteMessage**(*MessageNumber*)

Parameters

MessageNumber

Number of message to delete from the server. This value must be greater than zero. The first message in the mailbox is message number one.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **DeleteMessage** method only flags the message for deletion. The message is not actually removed from the mailbox until the client disconnects from the server, however it will no longer be accessible by the client. To prevent deleted messages from actually being removed from the mailbox, call the **Reset** method.

See Also

Reset Method

Disconnect Method

Terminate the connection with a server.

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

IsConnected Property, Connect Method, OnDisconnect Event

GetHeader Method

Returns the value of a header field from the specified message.

Syntax

object.**GetHeader**(*MessageNumber*, *HeaderField*, *HeaderValue*)

Parameters

MessageNumber

Number of message to retrieve header value from. This value must be greater than zero. The first message in the mailbox is message number one.

HeaderField

A string which specifies the message header to retrieve. The colon should not be included in this string.

HeaderValue

A string variable which will contain the value of the specified message header if the method is successful.

Return Value

A value of true is returned if the header was present and could be retrieved, otherwise a value of false is returned.

Remarks

The **GetHeader** method returns the value of a header field from the specified message. This allows an application to be able to easily determine the value of a header such as the sender, or the subject of the message. Any header field, including non-standard extensions, may be returned by this method.

This method uses the XTND XLST command, which is an extension to the POP3 protocol. If this command is not supported by the server, the method will attempt to retrieve the entire message header and return the value for the specified header field. This enables an application to use this method even if the server does not support command extensions.

See Also

HeaderField Property, HeaderValue Property, GetMessage Method

GetHeaders Method

Retrieves the headers for the specified message from the server.

Syntax

object.**GetHeaders**(*MessageNumber*, *Headers*)

Parameters

MessageNumber

Number of the message to retrieve from the server. This value must be greater than zero. The last available message number is returned by the **LastMessage** property.

Headers

A string or byte array which will contain the data transferred from the server when the method returns.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **GetHeaders** method is used to retrieve an message header block from the server and copy it into a local buffer. This method will cause the current thread to block until the article transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

GetMessage Method, OpenMessage Method, OnProgress Event

GetMessage Method

Retrieve a message from the server.

Syntax

object.**GetMessage**(*MessageNumber*, *Message*, [*Options*])

Parameters

MessageNumber

Number of message to retrieve from the server. This value must be greater than zero. The first message in the mailbox is message number one and the last message can be determined by checking the value of the **LastMessage** property.

Message

A string or byte array which will contain the data transferred from the server when the method returns.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **GetMessage** method is used to retrieve a message from the server and copy it into a local buffer. This method will cause the current thread to block until the message transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

DeleteMessage Method, GetHeaders Method, OpenMessage Method, OnProgress Event

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 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 popClient = CreateObject("SocketTools.PopClient.11")

nError = popClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize Method

OpenMessage Method

Open a message on the server.

Syntax

object.**OpenMessage**([*MessageNumber*], [*Options*])

Parameters

MessageNumber

Number of message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one. If this argument is omitted, the current message selected by the **Message** property will be opened.

Options

An optional integer value which specifies one or more options. This argument is reserved for future use and should be omitted.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **OpenMessage** method opens a message for reading from the server. The client can read the contents of the message using the **Read** method, and once all of the data has been read, the message should be closed by calling the **CloseMessage** method.

See Also

Message Property, CloseMessage Method, GetMessage Method, Read Method

Read Method

Return data read from the server.

Syntax

object.**Read**(*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

The number of bytes actually read from the server 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 server, 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.

See Also

IsConnected Property, IsReadable Property, Write Method, OnRead Event, OnWrite 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. Calling this method will also prevent any messages which have been marked for deletion from being removed from the mailbox.

See Also

Cancel Method, Initialize Method, Uninitialize Method

SendMessage Method

Submits the contents of a specified file to the mail server for delivery.

Syntax

object.**SendMessage**(*Message*, [*Options*])

Parameters

Message

A string or byte array which contains the message to be delivered. The To:, Cc: and Bcc: header fields will be scanned for recipient addresses, and the Bcc: line will be deleted before the message is delivered.

Options

An optional integer value which specifies one or more options. This argument is reserved for future use and should be omitted.

Return Value

A value of zero is returned if the message was submitted successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **SendMessage** method submits the messages to the mail server for delivery. The message format must comply with the RFC 822 standard, with the header and body separated by a blank line, and each line terminated with carriage-return/linefeed characters.

This method will cause the current thread to block until the message transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

This method requires that the server support the XTND XMIT command. Although using this method to send mail has the advantage that the sender is authenticated (because the user must first login to the server), it is not widely supported. For general purpose mail delivery service, it is recommended that an application use the Simple Mail Transfer Protocol (SMTP).

See Also

StoreMessage Method

StoreMessage Method

Retrieve a message from the server and store it in a local file.

Syntax

object.**StoreMessage**(*MessageNumber*, *FileName*)

Parameters

MessageNumber

An integer that specifies the message to retrieve. This value must be greater than zero. The first message in the mailbox is message number one.

FileName

A string which specifies the file that the message will be stored in. If an empty string is passed as an argument, the message is copied to the system clipboard.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **StoreMessage** method retrieves a message from the server and stores it in a file on the local system. The contents of the message is stored as a text file, using the specified file name. This method will cause the current thread to block until the message transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

See Also

GetMessage Method, OpenMessage 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. 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 server.

Syntax

object.**Write**(*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 because 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 you wish to send the data without conversion, use a **Byte** array as the buffer instead of a **String** variable.

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 server, or -1 if an error was encountered.

Remarks

The **Write** method sends the data in ***buffer*** to the server. 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 blocking, the application will wait until the data can be sent. If the control is non-blocking and the write fails because it could not send all of the data to the server, the **OnWrite** event will be fired when the server can accept data again.

See Also

IsConnected Property, IsWritable Property, Timeout Property, Read Method, OnWrite Event

Post Office Protocol Control Events

Event	Description
OnCancel	This event is generated when a blocking operation is canceled
OnCommand	This event is generated when the server processes a command issued by the client
OnConnect	This event is generated when a connection is established
OnDisconnect	This event is generated when a connection is terminated
OnError	This event is generated when a control error occurs
OnProgress	This event is generated during data transfer
OnRead	This event is generated when data is available to be read
OnTimeout	This event is generated when a blocking operation times out

OnCancel Event

The **OnCancel** event is generated when a blocking operation is canceled.

Syntax

Sub *object*_**OnCancel** ([*Index* **As Integer**])

Remarks

This event is generated 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, OnTimeout Event

OnCommand Event

The **OnCommand** event is generated when the client sends a command to the server and receives a reply indicating the results of that command.

Syntax

Sub *object*_**OnCommand**([*Index* **As Integer**], **ByVal** *ResultCode* **As Variant**, **ByVal** *ResultString* **As Variant**)

Remarks

The **OnCommand** event is generated when the client receives a reply from the server after some action has been taken. If the ***ResultCode*** argument has a value of true, that corresponds to an OK response from the server which indicates that the command was successful. If the argument has a value of false, that corresponds to an ERR response from the server which indicates that the command failed.

The ***ResultString*** argument contains the descriptive string returned by the server which describes the result. The string contents may vary depending on the type of server.

See Also

ResultCode Property, ResultString Property, Command Method

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 server as a result of a **Connect** method call. This event is only triggered when the **Blocking** property is set to False.

See Also

Blocking Property, Connect Method, OnDisconnect Event, OnWrite 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. This event is only triggered when the **Blocking** property is set to False.

When the **OnDisconnect** event fires, it is possible that there may still be buffered data available to read from the server. Before disconnecting from the server, the application should attempt to read any remaining data until the **Read** method returns a value of zero, or returns an error indicating that the operation would block.

See Also

Blocking Property, IsConnected Property, IsReadable Property, Connect Method, Disconnect Method, Read Method, 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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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 during data transfer.

Syntax

Sub *object*_**OnProgress** ([*Index* **As Integer**], **ByVal** *MessageNumber* **As Variant**, **ByVal** *MessageSize* **As Variant**, **ByVal** *MessageCopied* **As Variant**, **ByVal** *Percent* **As Variant**)

Remarks

The **OnProgress** event is generated during the transfer of data between the client and server, indicating the amount of data exchanged. For transfers of 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:

MessageNumber

The number of the message that is being retrieved.

MessageSize

The size of the file being transferred in bytes. This value may be zero if the control cannot obtain the size of the file from the server.

MessageCopied

The number of bytes that have been transferred between the client and server.

Percent

The percentage of data that's been transferred, expressed as an integer value between 0 and 100, inclusive. If the size of the file on the server cannot be determined, this value will always be 100.

Note that this event is only generated when message data is retrieved using the **GetHeaders** or **GetMessage** method. If the client is reading the message data directly from the server using the **Read** method, the application is responsible for calculating the completion percentage and updating any user interface controls.

See Also

GetHeaders Method, GetMessage 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 triggered when the **Blocking** property is set to False.

See Also

IsReadable Property, 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. This event is only triggered when the **Blocking** property is set to True.

See Also

Timeout Property, OnCancel Event

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 triggered when the **Blocking** property is set to False.

See Also

IsWritable Property, Read Method, Write Method, OnConnect Event, OnRead Event

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	CSRASX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.Dialer.11
ClassID	CF645AD7-3F40-4C1A-8E8A-ABCA925A4BF7
Threading Model	Apartment
Help File	CSW11HLP.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 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 compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the

latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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	Enable or disable error handling by the container of the control
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 object

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
Primary DNS nameserver IP address	
Alternate DNS nameserver IP address	
Primary WINS nameserver IP address	
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

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** [= { True | False }]

Remarks

Error handling for 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, methods will not raise an exception if an error occurs. Instead, the application should check the return value of the method and report any errors based on that value. It is the responsibility of the application to interpret the error code and take an appropriate action. This is the default value for the property.

If the **ThrowError** property is set to True, any method which generates an error will cause the component to raise an exception which must be handled or the application will terminate.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of this property. This property only controls how errors are handled when calling methods.

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 object.

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 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 is generated when a connection is established
OnDisconnect	This event is generated when a connection is terminated
OnError	This event is generated when a control error occurs
OnStatus	This event is generated when the control state changes
OnTimeout	This event is generated 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 is generated 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 is generated 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 is generated 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

Remote Command Control

Execute commands on a server or establish an interactive terminal session.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	RshClientCtl.RshClient
File Name	CSRSHX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.RshClient.11
ClassID	D82CEE60-9C78-4F37-BD5A-E8A34B438AD9
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 1282

Overview

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

This control should not be used when connecting to a server over the Internet because the user credentials are sent as unencrypted text. For secure remote command execution and interactive terminal sessions, it is recommended that you use the SocketTools Secure Shell (SSH) control instead.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

Remote Command Control Properties

Property	Description
AutoResolve	Determines if host names and IP addresses are automatically resolved
Blocking	Gets and sets the blocking state of the control
CodePage	Gets and sets the code page used when reading and writing text
Command	Gets and sets the command to be executed on the server
HostAddress	Gets and sets the IP address of the server
HostName	Gets and sets the name of the server
IsBlocked	Return if the control is blocked performing an operation
IsConnected	Determine if the control is connected to a server
IsInitialized	Determine if the control has been initialized
IsReadable	Return if data can be read from the server without blocking
IsWritable	Return if data can be sent to the server without blocking
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error to occur
Password	Gets and sets the password for the current user
RemotePort	Gets and sets the port number for a remote connection
Terminal	Gets and sets the terminal type used by the control
ThrowError	Enable or disable error handling by the container of the control
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
UserName	Gets and sets the current user name
Version	Return the current version of the object

AutoResolve Property

Determines if host names and IP 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 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 **AutoResolve** to False.

Data Type

Boolean

See Also

HostAddress Property, HostName Property

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 read data when none has been written, an error is generated. Events such as **OnConnect**, **OnDisconnect**, **OnRead** and **OnWrite** are only fired if the connection is non-blocking.

Data Type

Boolean

See Also

IsBlocked Property, IsReadable Property, IsWritable 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
Text sent and received using a string should be converted using the ANSI code page for the current locale. This is the default encoding type.	
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. It is not recommended that you use this code page unless you know that the remote host is sending text which includes OEM characters.	
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.	
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.	
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.	
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 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, Write Method

Command Property

Gets and sets the command to be executed on the server.

Syntax

object.**Command** [= *command*]

Remarks

The **Command** property sets the default command string that will be sent to the server when the **Execute** method is called and no command is explicitly specified as an argument.

Data Type

String

See Also

Password Property, RemotePort Property, UserName Property, Execute Method

HostAddress Property

Gets and sets the IP address of the server.

Syntax

object.**HostAddress** [= *ipaddress*]

Remarks

The **HostAddress** property can be used to set the IP address for a server 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, HostName Property

HostName Property

Gets and sets the name of the server.

Syntax

object.**HostName** [= *hostname*]

Remarks

The **HostName** property should be set to the name of the server 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

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.

Note that this property will return True if there is ***any*** blocking operation being performed by the application, regardless if the specified control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

Blocking Property, LastError Property

IsConnected Property

Determine if the control is connected to a server.

Syntax

object.**IsConnected**

Remarks

The **IsConnected** read-only property is set to a value of true if the control is connected with a server, otherwise the property has a value of false.

Data Type

Boolean

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

IsReadable Property

Return if data can be read from the server without blocking.

Syntax

object.**IsReadable**

Remarks

The **IsReadable** property returns True if data can be read from the server without blocking. For non-blocking connections, this property can be checked before the application attempts to read the data, preventing an error.

Data Type

Boolean

See Also

IsConnected Property, Read Method, OnRead Event

IsWritable Property

Return if data can be sent to the server without blocking.

Syntax

object.**IsWritable**

Remarks

The **IsWritable** property returns True if data can be written without blocking. For non-blocking connections, this property can be checked before the application attempts to send data to the server, 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 send data without an error. The next operation may result in an **stErrorOperationWouldBlock** or **stErrorOperationInProgress** error. The application must treat these errors as recoverable, and should be prepared to retry operations that result in them.

Data Type

Boolean

See Also

IsReadable Property, Write Method, OnWrite Event

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, OnError Event

LastErrorString Property

Return a description of the last error to occur

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

Password Property

Gets and sets the password for the current user.

Syntax

object.**Password** [= *password*]

Remarks

The **Password** property specifies the password used to authenticate the user. This property is used as the default value for the **Connect** method if no password is specified as an argument.

Data Type

String

See Also

UserName Property, Execute Method, Login Method

RemotePort Property

Gets and sets the port number for a remote connection.

Syntax

object.**RemotePort** [= *portnumber*]

Remarks

The **RemotePort** property is used to set the port number that the control will use to establish a connection with the server.

The following values are the default port numbers used by the control:

Value	Description
rshPortExec	A connection is established with the server using port 512, the rexec service. This service requires that the client provide a username and password to execute the specified command.
rshPortLogin	A connection is established with the server using port 513, the rlogin service. This service is similar Telnet in that it provides an interactive login session. The client is provided with a command prompt and can enter commands which are executed on the server.
rshPortRshell	A connection is established with the server using port 514, the rshell service. This service uses host equivalence to authenticate the user. With host equivalence, the server considers the client to be equivalent to itself, and as long as the specified user exists on the server, the client is permitted to execute commands on behalf of the user without requiring a password. Host equivalence is configured by the server administrator.

Data Type

Integer (Int32)

See Also

HostAddress Property, HostName Property, Execute Method, Login Method

Terminal Property

Gets and sets the terminal type used by the control.

Syntax

object.**Terminal** [= *termtype*]

Remarks

The **Terminal** property specifies the terminal type of the server for display purposes. On UNIX based systems, the terminal name corresponds to a termcap or terminfo entry as set in the TERM environment variable. On Windows based systems which implement the rlogin service, this property may be ignored and the server will assume that the client is capable of displaying ANSI escape sequences. On VMS systems, the terminal name should correspond to the terminal type used with the SET TERMINAL/DEVICE command.

If this property is set to an empty string and no terminal type is specified when the **Login** method is called, a default terminal type named "unknown" will be used. On most UNIX and VMS systems this defines a terminal which is not capable of cursor positioning using control or escape sequences. This terminal type may not be recognized and an error may be displayed when the user logs in indicating that the terminal type is invalid.

Refer to the documentation for the server system to determine what terminal type names are available to you. Remember that on UNIX systems, the terminal type is case-sensitive. Some of the more common terminal types are:

Terminal Type	Description
ansi	This terminal type is usually available on UNIX based servers. This specifies that the client is capable of displaying standard ANSI escape sequences for cursor control.
dumb	This terminal type typically specifies a terminal display which does not support control or escape sequences for cursor positioning. If you do not want escape sequences embedded in the data stream and the server returns an error if the terminal type is not specified, try using this terminal type.
pcansi	This terminal type is usually available on UNIX based servers. This specifies that the client is a using a PC terminal emulator that supports basic ANSI escape sequences for cursor control. This may also enable escape sequences which can set the display colors.
vt100	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT100. This specifies that the client is capable of emulating a DEC VT100 terminal. The VT100 supports many of the same cursor control sequences as an ANSI terminal.
vt220	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT220. This specifies that the client is capable of emulating a DEC VT220 terminal, which is a later version of the VT100.
vt320	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT320.

	This specifies that the client is capable of emulating a DEC VT320 terminal, which is similar to the VT100 and VT220 and provides advanced features such as the ability to set display colors.
xterm	This terminal type is may be available on UNIX based servers which have X Windows installed. This specifies that the client is a using the X Windows xterm emulator which supports standard ANSI escape sequences for cursor control.

Data Type

String

See Also

UserName Property, Login Method

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
RshClient1.ThrowError = False
nError = RshClient1.Connect(strHostName)

If nError > 0 Then
    MsgBox RshClient1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

RshClient1.ThrowError = True
RshClient1.Connect strHostName

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error.

Note that the **Timeout** property also determines the amount of time the control will spend attempting to connect to a server. If a connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
controlTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
controlTraceError	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.
controlTraceWarning	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.
controlTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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

UserName Property

Gets and sets the current user name.

Syntax

object.**UserName** [= *username*]

Remarks

The **UserName** property specifies the user that is logging in to the server, and is required for authentication purposes. This property is used as the default value for the **Connect** method if no password is specified as an argument.

Data Type

String

See Also

Password Property, Execute Method, Login Method

Version Property

Return the current version of the object.

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

Remote Command Control Methods

Method	Description
Cancel	Cancels the current blocking network operation
Disconnect	Terminate the connection with a server
Execute	Execute the specified command on the server
Initialize	Initialize the control and validate the runtime license key
Login	Establish an interactive terminal session for the specified user
Read	Return data read from the server
Reset	Reset the internal state of the control
Search	Search for a specific character sequence in the data stream
SendKey	Send a key code to the server
Uninitialize	Uninitialize the control and release any system resources that were allocated
Write	Write data to the server

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

Disconnect Method, Reset Method, OnCancel Event

Disconnect Method

Terminate the connection with a server.

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

IsConnected Property, Execute Method, Login Method, OnDisconnect Event

Execute Method

Execute the specified command on the server.

Syntax

object.**Execute**([*RemoteHost*], [*RemotePort*], [*UserName*], [*Password*], [*Command*], [*Timeout*], [*Options*])

Parameters

RemoteHost

A optional string argument which specifies the name of the server to connect to. The string may either be an IP address or a fully qualified domain name. If the argument is omitted, the value of the **HostAddress** or **HostName** property will be used.

RemotePort

An optional integer argument which specifies the port number to connect to. This method may be used to either connect to the rexec service or the rshell service, and which service is selected depends on the port number provided. If this argument is omitted, the value of the **RemotePort** property will be used.

Value	Description
rshPortExec	A connection is established with the server using port 512, the rexec service. This service requires that the client provide a username and password to execute the specified command.
rshPortRshell	A connection is established with the server using port 514, the rshell service. This service uses host equivalence to authenticate the user. With host equivalence, the server considers the client to be equivalent to itself, and as long as the specified user exists on the server, the client is permitted to execute commands on behalf of the user without requiring a password. Host equivalence is configured by the server administrator.

UserName

An optional string argument which specifies the username which used to authenticate the client session. If this argument is omitted, the value of the **UserName** property will be used.

Password

An optional string argument which specifies the password to be used to authenticate the user. If this argument is omitted, the value of the **Password** property will be used. A password is only used if the client is connecting to the rexec service. The rshell service uses host equivalence to authenticate the user and this argument will be ignored.

Command

An optional string argument which specifies the command to be executed on the server. If this argument is omitted, the value of the **Command** property will be used.

Timeout

An optional integer argument which specifies the number of seconds that the client will wait for a response from the server. If this argument is omitted, the value of the **Timeout** property will be used. This value is only used for synchronous connections where the **Blocking** property is set to True.

Options

An optional integer argument which specifies one or more connection options. This argument is reserved for future use and should be omitted.

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 **Execute** method executes the specified command on a server. Output from the command may be read using the **Read** method. Input can be supplied to the program using the **Write** method. To search for a specific sequence of bytes in the output returned by the server, use the **Search** method.

See Also

Disconnect Method, Login Method, Read Method, Search 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 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 rshClient = CreateObject("SocketTools.RshClient.11")

nError = rshClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize Method

Login Method

Establish an interactive terminal session for the specified user.

Syntax

object.**Login**([*RemoteHost*], [*RemotePort*], [*UserName*], [*Timeout*], [*Options*])

Parameters

RemoteHost

A optional string argument which specifies the name of the server to connect to. The string may either be an IP address or a fully qualified domain name. If the argument is omitted, the value of the **HostAddress** or **HostName** property will be used.

RemotePort

An optional integer argument which specifies the port number to connect to. If this argument is omitted, the value of the **RemotePort** property will be used. A remote port of zero specifies that port 513 should be used, which is the standard port number for the rlogin service.

UserName

An optional string argument which specifies the username which should be used to authenticate the client session. If this argument is omitted, the value of the **UserName** property will be used.

Timeout

An optional integer argument which specifies the number of seconds that the client will wait for a response from the server. If this argument is omitted, the value of the **Timeout** property will be used. This value is only used for synchronous connections where the **Blocking** property is set to True.

Options

An optional integer argument which specifies one or more connection options. This argument is reserved for future use and should be omitted.

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 **Login** method logs the specified user in on the server. Note that no password is provided to the server. This is because the remote login service uses ***user equivalence***. If the client system is recognized by the server as being equivalent, the login will proceed directly. If the client system is not recognized, the server will prompt the user for a password. For more information about user equivalence and the remote login service, refer to your server's operating system documentation.

Output from the command may be read using the **Read** method. Input can be supplied to the program using the **Write** method. To search for a specific sequence of bytes in the output returned by the server, use the **Search** method.

See Also

Disconnect Method, Execute Method, Read Method, Search Method

Read Method

Return data read from the server.

Syntax

object.**Read**(*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.

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 server 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 server, 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 stop until data is returned by the server or the connection is closed. Note that it is possible for the returned data to contain embedded null characters.

See Also

IsConnected Property, IsReadable Property, Write Method, OnRead Event, OnWrite 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

Search Method

Search for a specific character sequence in the data stream.

Syntax

object.**Search**(*String*, [*Buffer*], [*Length*], [*Options*])

Parameters

String

A string argument which specifies the sequence of characters to search for in the data stream. When the control encounters this sequence, the method will return.

Buffer

An optional string or byte array buffer that will contain the output sent by the server, up to and including the search string character sequence. If this argument is omitted, the control will still search for the character sequence but any output sent by the server will be discarded.

Length

An optional integer value which specifies the maximum number of bytes of data to store in the buffer. If this argument is omitted, no limit will be placed on the amount of output buffered by the control.

Options

An optional integer argument which is reserved for future use. This argument should be omitted.

Return Value

This method returns a Boolean value. A return value of true indicates that the search string was found in the data stream. A return value of False indicates that the search string was not found in the amount of time specified by the **Timeout** property or that the server closed the connection.

Remarks

The **Search** method searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server, such as executing a command and processing the output. The function collects the output from the server and stores it in a buffer provided by the caller. When the function returns, the buffer will contain everything sent by the server up to and including the search string.

See Also

IsReadable Property, Timeout Property, Execute Method, Login Method, Read Method

SendKey Method

Send a key code to the server.

Syntax

object.**SendKey**(*Key*)

Parameters

Key

A value which specifies the key code to send to the server. This may be a single byte, in which case it is sent to the server as-is. If a numeric value is specified, then this is considered to be an ASCII character value and it is sent to the server as a single byte. The value must be between 1 and 255. If the key code value is 0, then the method returns without sending any data. If the value is greater than 255, an error will be raised. If the *Key* argument is a string, then the method will send that string to the server. An empty string is ignored and the method will return without sending any data. An error will be returned if the string is longer than 128 bytes.

Return Value

This method will return a value of true if the key code was successfully sent to the server. If the key cannot be sent, the method will return False and the **LastError** property will contain the error code that indicates the reason for the failure. This method will also return False if the key code value is zero or an empty string is passed by the caller.

Remarks

The **SendKey** method sends a key code to the server. This method is useful if the application needs to send a single character to the server, as opposed to using the **Write** method which should be used for sending large amounts of data.

The strings sent by the **SendKey** method are typically short escape sequences which are generated by a terminal emulator when the user presses a special key, such as a function key. For example, a DEC VT100 terminal sends the escape sequence <ESC>[M when the user presses the F1 function key. To simulate this, those three bytes could be passed as the *Key* value.

Example

The following example demonstrates how to use the **SendKey** method in conjunction with the **KeyMapped** and **KeyPress** events in the Terminal Emulator control:

```
Private Sub Terminal1_KeyMapped(KeyIndex As Integer, Shift As Integer, KeyString As
String)
    RshClient1.SendKey KeyString
End Sub

Private Sub Terminal1_KeyPress(KeyAscii As Integer)
    RshClient1.SendKey KeyAscii
End Sub
```

See Also

IsWritable Property, Write Method, OnWrite 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 server.

Syntax

object.**Write**(*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 bytes actually written to the server, or -1 if an error was encountered.

Remarks

The **Write** method sends the data in ***buffer*** to the server. 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 blocking, the application will wait until the data can be sent. If the control is non-blocking and the write fails because it could not send all of the data to the server, the **OnWrite** event will be fired when the server can accept data again.

See Also

IsConnected Property, IsWritable Property, Timeout Property, Read Method, SendKey Method, OnWrite Event

Remote Command Control Events

Event	Description
OnCancel	This event is generated when a blocking operation is canceled
OnConnect	This event is generated when a connection is established
OnDisconnect	This event is generated when a connection is terminated
OnError	This event is generated when a control error occurs
OnRead	This event is generated when data is available to be read
OnTimeout	This event is generated when a blocking operation times out
OnWrite	This event is generated when data can be written to the server

OnCancel Event

The **OnCancel** event is generated when a blocking operation is canceled.

Syntax

Sub *object*_**OnCancel** ([*Index* **As Integer**])

Remarks

This event is generated 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, OnTimeout 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 server as a result of a **Connect** method call. This event is only triggered when the **Blocking** property is set to False.

See Also

Blocking Property, Execute Method, Login Method, OnDisconnect Event, OnWrite 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. This event is only triggered when the **Blocking** property is set to False.

When the **OnDisconnect** event fires, it is possible that there may still be buffered data available to read from the server. Before disconnecting from the server, the application should attempt to read any remaining data until the **Read** method returns a value of zero, or returns an error indicating that the operation would block.

See Also

Blocking Property, IsConnected Property, IsReadable Property, Disconnect Method, Read Method, 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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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

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 triggered when the **Blocking** property is set to False.

See Also

IsReadable Property, 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. This event is only triggered when the **Blocking** property is set to True.

See Also

Timeout Property, OnCancel Event

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 triggered when the **Blocking** property is set to False.

See Also

IsWritable Property, Read Method, SendKey Method, Write Method, OnConnect Event, OnRead Event

Simple Mail Transfer Protocol Control

Submit email messages for delivery to one or more recipients.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	SmtpClientCtl.SmtpClient
File Name	CSMTPX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.SmtpClient.11
ClassID	87388321-699D-4E44-8088-E6D6F7ACB8E8
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 821, RFC 1425, RFC 1869, RFC 2821

Overview

The Simple Mail Transfer Protocol (SMTP) enables applications to deliver email messages to one or more recipients. The control provides an interface for addressing and delivering messages, and extended features such as user authentication and delivery status notification. Unlike Microsoft's Messaging API (MAPI) or Collaboration Data Objects (CDO), there is no requirement to have certain third-party email applications installed or specific types of servers installed on the local system. The control can be used to deliver mail through a wide variety of systems, from standard UNIX based mail servers to Windows systems running Exchange or Lotus Notes and Domino.

Using this control, messages can be delivered directly to the recipient, or they can be routed through a relay server, such as an Internet service provider's mail system. The SocketTools MailMessage control can be integrated with this library in order to provide an extremely simple, yet flexible interface for composing and delivering messages.

This control supports secure connections using the standard SSL and TLS protocols. Both implicit and explicit TLS connections are supported, as well as client certificates used for authentication.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a

minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

Simple Mail Transfer Protocol Control Properties

Property	Description
AuthType	Gets and sets the method used to authenticate the user
AutoResolve	Determines if host names and IP addresses are automatically resolved
BearerToken	Gets and sets the OAuth 2.0 bearer token used for authentication
Blocking	Gets and sets the blocking state of the control
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 client certificate
CertificatePassword	Gets and sets the password associated with the client certificate
CertificateStatus	Return the status of the server certificate
CertificateStore	Gets and sets the name of the client 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
CurrentDate	Return the current date in the standard format used by email messages
Extended	Enable support for extended SMTP commands
HashStrength	Return the length of the message digest that was selected
HostAddress	Gets and sets the IP address of the server
HostName	Gets and sets the name of the server
IsBlocked	Return if the control is blocked performing an operation
IsConnected	Determine if the control is connected to a server
IsInitialized	Determine if the control has been initialized
IsWritable	Return if data can be sent to the server without blocking
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error to occur
LocalDomain	Gets and sets the local domain name
Options	Gets and sets the options that are used in establishing a connection
Password	Gets and sets the password for the current user
RemotePort	Gets and sets the port number for a remote connection
ResultCode	Return the result code of the previous action
ResultString	Return a string describing the results of the previous action
ReturnReceipt	Enable or disable delivery status notification
Secure	Set or return if a connection to the server is secure

SecureCipher	Return the encryption algorithm used to establish the secure connection with the server
SecureHash	Return the message digest selected when establishing the secure connection with the server
SecureKeyExchange	Return the key exchange algorithm used to establish the secure connection with the server
SecureProtocol	Gets and sets the security protocol used to establish the secure connection with the server
ThrowError	Enable or disable error handling by the container of the control
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
UserName	Gets and sets the current user name
Version	Return the current version of the object

AuthType Property

Gets and sets the method used to authenticate the user.

Syntax

object.**AuthType** [= *type*]

Remarks

The **AuthType** property specifies the type of authentication that should be used when the client connects to the mail server. The following authentication methods are supported:

Value	Description
smtpAuthLogin	The client will authenticate using the AUTH LOGIN command. This encodes the username and password, however the credentials are not encrypted and it is recommended you use a secure connection. This is the default method accepted by most mail servers and is the preferred authentication type for most clients.
smtpAuthPlain	The client will authenticate using the AUTH PLAIN command. This encodes the username and password, however the credentials are not encrypted and it is recommended you use a secure connection. The server must support the PLAIN Simple Authentication and Security Layer (SASL) mechanism as defined in RFC 4616.
smtpAuthXOAuth2	The client will authenticate using the AUTH XOAUTH2 command. This authentication method does not require the user password, instead the **BearerToken** property must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired or the user authentication will fail.
smtpAuthBearer	The client will authenticate using the AUTH OAUTHBEARER command as defined in RFC 7628. This authentication method does not require the user password, instead the **BearerToken** property must specify the OAuth 2.0 bearer token issued by the service provider. The application must provide a valid access token which has not expired or the user authentication will fail.

Data Type

Integer (Int32)

Remarks

The default authentication method is **smtpAuthLogin** and this is accepted by most mail servers. If you attempt to use an authentication method which is not supported by the server, the **Authenticate** method will fail and the last error code will be set to **stErrorInvalidAuthenticationType**.

All authentication methods require the mail server to support the standard service extensions for authentication as specified in the RFC 4954. The server must support the ESMTP protocol extensions and the AUTH command. A user name and password are required for authentication. If you wish to authenticate without a user password, you must use one of the OAuth 2.0 authentication methods.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an access token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application

and then requesting the access token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

The **smtpAuthXOAuth2** and **smtpAuthBearer** authentication methods are similar, but they are not interchangeable. Both use an OAuth 2.0 bearer token to authenticate the client session, but they differ in how the token is presented to the server. It is currently preferable to use the XOAUTH2 method because it is more widely available and some service providers do not yet support the OAUTHBEARER method.

Changing the value of the **BearerToken** property will automatically set the current authentication method to use OAuth 2.0.

See Also

BearerToken Property, Password Property, UserName Property, Authenticate Method, Connect Method

AutoResolve Property

Determines if host names and IP 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 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 **AutoResolve** to False.

Data Type

Boolean

See Also

HostAddress Property, HostName Property

BearerToken Property

Gets and sets the OAuth 2.0 bearer token for the current user.

Syntax

object.**BearerToken** [= *token*]

Remarks

The **BearerToken** property specifies the OAuth 2.0 bearer token used to authenticate the user. This property is used as the default value for the **Authenticate** method if the token is not provided as an parameter.

Assigning a value to this property will change the current authentication method to use OAuth 2.0 if necessary.

You should only use an OAuth 2.0 authentication method if you understand the process of how to request the access token. Obtaining an bearer token requires registering your application with the mail service provider (e.g.: Microsoft or Google), getting a unique client ID associated with your application and then requesting the token using the appropriate scope for the service. Obtaining the initial token will typically involve interactive confirmation on the part of the user, requiring they grant permission to your application to access their mail account.

Your application should not store an OAuth 2.0 bearer token for later use. They have a relatively short lifespan, typically about an hour, and are designed to be used with that session. You should specify offline access as part of the OAuth 2.0 scope if necessary and store the refresh token provided by the service. The refresh token has a much longer validity period and can be used to obtain a new bearer token when needed.

If the current authentication method does not use OAuth 2.0, this property will return an empty string and you should check the value of the **Password** property to obtain the current user's password. Refer to the **AuthType** property for more information on the available authentication methods.

Data Type

String

See Also

AuthType Property, Password Property, UserName Property, Authenticate Method, Connect 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 read data when none has been written, an error is generated. Events such as **OnConnect, OnDisconnect** and **OnWrite** are only fired if the connection is non-blocking.

Data Type

Boolean

See Also

IsBlocked Property, IsWritable Property

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 = SmtpClient1.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 client certificate.

Syntax

object.**CertificateName** [= *name*]

Remarks

This property sets the common name or friendly name of the certificate that should be used to establish the connection with the server. 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. If a certificate name is specified, the certificate must have a private key associated with it, otherwise the connection attempt 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 client 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
stCertificateNone	No certificate information is available. A secure connection was not established with the server.
stCertificateValid	The certificate is valid.
stCertificateNoMatch	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.
stCertificateExpired	The certificate has expired and is no longer valid. The client can examine the **CertificateExpires** property to determine when the certificate expired.
stCertificateRevoked	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.
stCertificateUntrusted	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.
stCertificateInvalid	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:

```
SmtpClient1.HostName = strHostName
SmtpClient1.Secure = True
```

```
nError = SmtpClient1.Connect()
If nError > 0 Then
     MsgBox "Unable to connect to server " & strHostName, vbExclamation
     Exit Sub
End If

If SmtpClient1.CertificateStatus <> stCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          SmtpClient1.Disconnect
          Exit Sub
     End If
End If

SmtpClient1.Disconnect
```

See Also

CertificateExpires Property, CertificateIssued Property, CertificateIssuer Property, CertificateSubject Property, Secure Property

CertificateStore Property

Gets and sets the name of the client certificate store or file.

Syntax

object.**CertificateStore** [= *store*]

Remarks

This property sets the name of the certificate store that contains the client certificate that should be used when establishing a secure connection with 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 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

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 = SmtpClient1.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 client certificate.

Syntax

object.**CertificateUser** [= *username*]

Remarks

This property sets the name of the user that owns the client certificate that will be used to establish a secure connection with the server. 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

CurrentDate Property

Return the current date in the standard format used by email messages.

Syntax

object.**CurrentDate**

Remarks

The **CurrentDate** property returns the current date and time in a format that is commonly used in email messages.

Data Type

String

Extended Property

Enable support for extended SMTP commands.

Syntax

object.**Extended** [= { True | False }]]

Remarks

The **Extended** property determines if the control should attempt to use extended SMTP commands. This is required to support options such as authentication, delivery status notification and message sizing.

Note that this property should be set before establishing a connection with the server using the **Connect** method. If the server does not support extended (ESMTP) features, the property value will be reset to False after the connection has been made.

Data Type

Boolean

See Also

Connect 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 server.

Syntax

object.**HostAddress** [= *ipaddress*]

Remarks

The **HostAddress** property can be used to set the IP address for a server 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, HostName Property

HostName Property

Gets and sets the name of the server.

Syntax

object.**HostName** [= *hostname*]

Remarks

The **HostName** property should be set to the name of the server 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

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.

Note that this property will return True if there is ***any*** blocking operation being performed by the application, regardless if the specified control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

Blocking Property, LastError Property

IsConnected Property

Determine if the control is connected to a server.

Syntax

object.**IsConnected**

Remarks

The **IsConnected** read-only property is set to a value of true if the control is connected with a server, otherwise the property has a value of false.

Data Type

Boolean

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

IsWritable Property

Return if data can be sent to the server without blocking.

Syntax

object.**IsWritable**

Remarks

The **IsWritable** property returns True if data can be written without blocking. For non-blocking connections, this property can be checked before the application attempts to send data to the server, 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 send data without an error. The next operation may result in an **stErrorOperationWouldBlock** or **stErrorOperationInProgress** error. The application must treat these errors as recoverable, and should be prepared to retry operations that result in them.

Data Type

Boolean

See Also

Write 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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

LocalDomain Property

Gets and sets the local domain name.

Syntax

object.**LocalDomain** [= *domain*]

Remarks

The **LocalDomain** returns the local domain name used when the client identifies itself to the mail server. If this property is an empty string, then the control will attempt to automatically determine the appropriate domain name to use based on the system configuration. Setting this property will cause the control to use that value when identifying itself to the server.

This property should only be set if it is absolutely necessary. In most cases, it is preferable to leave this property undefined and allow the control to automatically determine the correct domain name to use. Setting an invalid domain name may cause the mail server to reject the connection.

Data Type

String

See Also

HostName Property

Options Property

Gets and sets the options that are used in establishing a connection.

Syntax

object.**Options** [= *options*]

Remarks

The **Options** property is an integer value which specifies one or more options. The value specified for this property will be used as the default options when connecting to the server. The property value is created by using a bitwise operator with one or more of the following values:

Value	Description
smtpOptionNone	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used and the client will not attempt to use extended (ESMTP) features of the protocol. Note that if the mail server requires authentication, the **smtpOptionExtended** option must be specified.
smtpOptionExtended	Extended SMTP commands should be used if possible. This option enables features such as authentication and delivery status notification. If this option is not specified, the library will not attempt to use any extended features. This option is automatically enabled if the connection is established on port 587 because submitting messages for delivery using this port typically requires client authentication.
&H400	smtpOptionTunnel	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection

		is established.
&H800	smtpOptionTrustedSite	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 the TLS protocol.
&H1000	smtpOptionSecureExplicit	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. This option initiates the secure session using the STLS command.
&H2000	smtpOptionSecureImplicit	This option specifies the client should attempt to establish a secure connection with the server. It should only be used when the server expects an implicit TLS connection or does not implement RFC 2595 where the STLS command is used to negotiate a secure connection with the server.
&H8000	smtpOptionSecureFallback	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.
&H40000	smtpOptionPreferIPv6	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.

Data Type

Integer (Int32)

See Also

Extended Property, Secure Property, Connect Method

Password Property

Gets and sets the password for the current user.

Syntax

object.**Password** [= *password*]

Remarks

The **Password** property specifies the password used to authenticate the user. This property is used as the default value for the **Authenticate** method if no password is specified as an argument.

Refer to the **AuthType** property for more information on the available authentication methods. If you are using the OAuth 2.0 authentication method, this property should not be set to the user's password. Instead, you should set the **BearerToken** property to the OAuth 2.0 access token issued by the mail service provider. Note that these access tokens can be much larger than your typical password and are only valid for a limited period of time.

You can use the **Password** property to specify an OAuth 2.0 bearer token. However, it is recommended that you use the **BearerToken** property instead of assigning it to this property. It will ensure compatibility with future versions of the control and make it clear in your code you are using an OAuth 2.0 bearer token and not a password. If the **AuthType** property specifies one of the OAuth 2.0 authentication methods, this property will return the bearer token.

Data Type

String

See Also

BearerToken Property, UserName Property, Authenticate Method, Connect Method

RemotePort Property

Gets and sets the port number for a remote connection.

Syntax

object.**RemotePort** [= *portnumber*]

Remarks

The **RemotePort** property is used to set the port number that the control will use to establish a connection with the server. For standard connections, the default port number is 25. An alternative port is 587, which is commonly used by authenticated clients to submit messages for delivery. For secure connections, the default port number is 465. If the secure port number is specified, an implicit TLS connection will be established by default.

Data Type

Integer (Int32)

See Also

HostAddress Property, HostName Property

ResultCode Property

Return the result code of the previous action.

Syntax

object.**ResultCode**

Remarks

The **ResultCode** read-only property returns the result code of the last action performed by the client. This property should be checked after the **Command** method is used to execute a command on the server to determine if the operation was successful. Result codes are three-digit numeric values returned by the server and may be broken down into the following ranges:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

It is important to note that while some result codes have become standardized, not all servers respond to commands using the same result codes. For example, one server may respond with a result code of 221 to indicate success, while another may respond with a value of 235. It is recommended that applications check for ranges of values to determine if a command was successful, not a specific value.

Data Type

Integer (Int32)

See Also

ResultString Property, Command Method, OnCommand Event

ResultString Property

Return a string describing the results of the previous action.

Syntax

object.**ResultString**

Remarks

The **ResultString** read-only property returns the result string from the last action taken by the client. This string is generated by the server, and typically is used to describe the result code. For example, if an error is indicated by the result code, the result string may describe the condition that caused the error.

Data Type

String

See Also

ResultCode Property, Command Method, OnCommand Event

ReturnReceipt Property

Enable or disable delivery status notification.

Syntax

object.**ReturnReceipt** [= { True | False }]]

Remarks

The **ReturnReceipt** property enables or disables delivery status notification (DSN) by the mail server. If the property is set to True, a mail message will be automatically returned to the sender indicating if the message was delivered successfully, unsuccessfully or delayed by the mail server. If the property is set to False, no delivery status information is sent back to the sender.

Note that delivery status notification is not available on all servers. It is also important to note that a message indicating that delivery was successful does not mean that the message was actually read by the recipient, only that it was delivered to their mailbox.

Data Type

Boolean

See Also

AddRecipient Method, CreateMessage Method

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 must 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 strongly recommended that any application that sets this property True use error handling to trap an errors that may occur. If the control 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.

Data Type

Boolean

Example

The following example establishes a secure connection to a server:

```
SmtpClient1.HostName = strHostName
SmtpClient1.RemotePort = 587
SmtpClient1.UserName = strUserName
SmtpClient1.Password = strPassword
SmtpClient1.Secure = True

nError = SmtpClient1.Connect()
If nError > 0 Then
    MsgBox "Unable to connect to server " & strHostName, vbExclamation
    Exit Sub
End If

If SmtpClient1.CertificateStatus <> stCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          SmtpClient1.Disconnect
          Exit Sub
     End If
End If
```

See Also

CertificateStatus 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
stCipherNone	No cipher has been selected. This is not a secure connection with the server.
stCipherRC2	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.
stCipherRC4	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.
stCipherRC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
stCipherDES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher using 56-bit keys.
stCipherDES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively using a 168-bit key length.
stCipherDESX	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.
stCipherAES	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.
stCipherSkipjack	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
stCipherBlowfish	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
stHashMD5	The MD5 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA1	The SHA-1 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA256	The SHA-256 algorithm has been selected.
stHashSHA384	The SHA-384 algorithm has been selected.
stHashSHA512	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
stKeyExchangeNone	No key exchange algorithm has been selected. This is not a secure connection with the server.
stKeyExchangeRSA	The RSA public key exchange algorithm has been selected.
stKeyExchangeKEA	The KEA public key exchange algorithm has been selected. This is an improved version of the Diffie-Hellman public key algorithm.
stKeyExchangeDH	The Diffie-Hellman public key exchange algorithm has been selected.
stKeyExchangeECDH	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

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
SmtpClient1.ThrowError = False
nError = SmtpClient1.Connect(strHostName)

If nError > 0 Then
    MsgBox SmtpClient1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

SmtpClient1.ThrowError = True
SmtpClient1.Connect strHostName

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error.

Note that the **Timeout** property also determines the amount of time the control will spend attempting to connect to a server. If a connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
smtpTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
smtpTraceError	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.
smtpTraceWarning	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.
smtpTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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

UserName Property

Gets and sets the current user name.

Syntax

object.**UserName** [= *username*]

Remarks

The **UserName** property specifies the user that is logging in to the server, and is required for authentication purposes. This property is used as the default value for the **Connect** method if no password is specified as an argument.

Data Type

String

See Also

Password Property, Connect Method

Version Property

Return the current version of the object.

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

Simple Mail Transfer Protocol Control Methods

Method	Description
AddRecipient	Add an address to the recipient list for the current message
AppendMessage	Append text to the current message being composed
Authenticate	Authenticate the client session
Cancel	Cancels the current blocking network operation
CloseMessage	Closes the current message
Command	Send a custom command to the server
Connect	Establish a connection with a server
CreateMessage	Begin the composition of a new message to be delivered
Disconnect	Terminate the connection with a server
ExpandAddress	Expand the specified email address
Initialize	Initialize the control and validate the runtime license key
Reset	Reset the internal state of the control
SendMessage	Send the specified message through the mail server
Uninitialize	Uninitialize the control and release any system resources that were allocated
VerifyAddress	Verify the specified email address
Write	Write data to the server

AddRecipient Method

Add an address to the recipient list for the current message.

Syntax

object.**AddRecipient**(*Address*)

Parameters

Address

A string value that specifies the email address of a recipient.

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 **AddRecipient** method adds the specified address to the recipient list for the current message. This method should be called after the message transaction has begun with a call to the **CreateMessage** method. Most servers impose a limit of approximately 100 recipient addresses that will be accepted for a single message.

See Also

AppendMessage Method, CloseMessage Method, CreateMessage Method

AppendMessage Method

Append text to the current message being composed.

Syntax

object.**AppendMessage**(*Message*, [*Options*])

Parameters

Message

A string or byte array which will contain the article to be posted to the server.

Options

An optional integer value which specifies one or more options. This argument is reserved for future use and should be omitted.

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 **AppendMessage** method appends the specified text to the current message. This method will cause the current thread to block until the article transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

This method is useful for composing a message where the application needs to dynamically create the header, followed by a large amount of text. The message contents should be text, with each line terminated with a carriage return and linefeed character. Not all mail servers support sending 8-bit characters, so the message contents may need to be encoded if it uses anything other than standard US ASCII. To append binary data, it must be encoded using either the uucode or base64 (MIME) algorithms. It is recommended that you use the Mail Message control to handle file attachments and other complex message types.

See Also

AddRecipient Method, CloseMessage Method, CreateMessage Method

Authenticate Method

Authenticate the client session.

Syntax

object.**Authenticate**([*UserName*], [*Password*])

Parameters

UserName

An optional string argument which specifies the username used to authenticate the client session. If the argument is omitted, the value assigned to the **UserName** property will be used instead.

Password

An optional string argument which specifies the password used to authenticate the client session. If the argument is omitted, the value assigned to the **Password** property will be used instead. If you are using OAuth 2.0 authentication, this parameter specifies the bearer token provided by the mail service.

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 **Authenticate** method is used to authenticate the current client session to the mail server, ensuring that only valid users may deliver messages through the server. This method uses the LOGIN authentication mechanism by default and you can specify an alternate authentication method by setting the the **AuthType** property.

Authentication requires the server to support the AUTH extended SMTP command and the **Extended** property must be set to True. If the server does not support the specified type of authentication, an error will be returned.

If you with to use OAuth 2.0 for authentication, set the **AuthType** property to the desired authentication type prior to calling this method. The connection must be secure, and the server must advertise its support for OAuth 2.0 or the authentication attempt will fail. This method will not attempt to automatically refresh an expired token.

See Also

AuthType Property, BearerToken Property, Password Property, UserName Property, Connect Method

Cancel Method

Cancels the current blocking network operation.

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

Disconnect Method, Reset Method, OnCancel Event

CloseMessage Method

Closes the current message.

Syntax

object.**CloseMessage**

Parameters

None.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CloseMessage** method closes the current message and completes the submission of the message to the server. This method is only required if the message is being submitted using the **AppendMessage** or **Write** methods.

See Also

AppendMessage Method, CreateMessage Method

Command Method

Send a custom command to the server.

Syntax

object.**Command**(*Command*, [*Parameters*], [*Options*])

Parameters

Command

A string which specifies the command to send. Valid commands vary based on the Internet protocol and the type of server that the client is connected to. Consult the protocol standard and/or the technical reference documentation for the server to determine what commands may be issued by a client application.

Parameters

An optional string which specifies one or more parameters to be sent along with the command. If more than one parameter is required, most Internet protocols require that they be separated by a single space character. Consult the protocol standard and/or technical reference documentation for the server to determine what parameters should be provided when issuing a specific command. If no parameters are required for the command, this argument may be omitted.

Options

A numeric value which specifies one or more options. Currently this argument is reserved and should either be omitted, or a value of zero should always be used.

Return Value

A value of zero is returned if the command was successful. Otherwise, a non-zero error code is returned which indicates the cause of the failure. To determine the result code returned by the server in response to the command, read the value of the **ResultCode** property.

Remarks

The **Command** method sends a command to the server and processes the result code sent back in response to that command. This method can be used to send custom commands to a server to take advantage of features or capabilities that may not be supported internally by the control.

See Also

ResultCode Property, ResultString Property, OnCommand Event

Connect Method

Establish a connection with a server.

Syntax

object.**Connect**([*RemoteHost*], [*RemotePort*], [*UserName*], [*Password*], [*Timeout*], [*Options*])

Parameters

RemoteHost

A string which specifies the host name or IP address of the server. If this argument is not specified, it defaults to the value of the **HostAddress** property if it is defined. Otherwise, it defaults to the value of the **HostName** property.

RemotePort

A number which specifies the port to connect to on the server. If this argument is not specified, it defaults to the value of the **RemotePort** property. A value of zero specifies that the default port number should be used. For standard connections, the default port number is 25. An alternative port is 587, which is commonly used by authenticated clients to submit messages for delivery. For implicit TLS connections, the default port number is 465.

UserName

An optional string argument which specifies the user name to be used with authentication. If this argument is not specified, it defaults to the value of the **UserName** property. Note that for authentication to be performed, the **Extended** property must be set to True.

Password

An optional string argument which specifies the password to be used with authentication. If this argument is not specified, it defaults to the value of the **Password** property. Note that for authentication to be performed, the **Extended** property must be set to True.

Timeout

The number of seconds that the client will wait for a response before failing the operation. If this argument is not specified, the value of the **Timeout** property will be used as the default.

Options

A numeric value which specifies one or more options. If this argument is omitted or a value of zero is specified, a default connection will be established. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description
smtpOptionNone	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used and the client will not attempt to use extended (ESMTP) features of the protocol. Note that if the mail server requires authentication, the **smtpOptionExtended** option must be specified.

smtpOptionExtended	Extended SMTP commands should be used if possible. This option enables features such as authentication and delivery status notification. If this option is not specified, the library will not attempt to use any extended features. This option is automatically enabled if the connection is established on port 587 because submitting messages for delivery using this port typically requires client authentication.	
&H400	smtpOptionTunnel	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
&H800	smtpOptionTrustedSite	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 the TLS protocol.
&H1000	smtpOptionSecureExplicit	This option specifies that a secure connection should be established with the server and requires that the server support the TLS protocol. This option initiates the secure session using the STLS command.
&H2000	smtpOptionSecureImplicit	This option specifies the client should attempt to establish a secure connection with the server. It should only be used when the server expects an implicit TLS connection or does not implement RFC 2595 where the STLS command is used to negotiate a secure connection

		with the server.
&H8000	smtpOptionSecureFallback	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.
&H40000	smtpOptionPreferIPv6	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 **Connect** method establishes a connection with the mail server. All arguments are optional. If a given argument is omitted, then the corresponding property values are used as defaults. Specifying a username and password only sets the default **UserName** and **Password** property values. If authentication is required, the client must explicitly call the **Authenticate** method after the connection has been established.

See Also

Extended Property, HostAddress Property, HostName Property, Options Property, RemotePort Property, Authenticate Method, Disconnect Method, OnConnect Event

CreateMessage Method

Begin the composition of a new message to be delivered.

Syntax

object.**CreateMessage**(*Sender*, [*MessageSize*])

Parameters

Sender

A string which specifies the email address of the user sending the message. This typically corresponds to the address in the From header of the message, but it is not required that they be the same.

MessageSize

An integer which specifies the size of the message in bytes. If the size of the message is unknown, this argument should be omitted or passed as value of zero. This argument is ignored if the server does not support extended features. If the message size is larger than what the server will accept, this method will fail. Most Internet Service Providers impose a limit on the size of an email message, typically between 5 and 10 megabytes.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **CreateMessage** method begins the composition of a new message to be submitted to the mail server for delivery. There are several steps that must be followed when dynamically composing a message using the **CreateMessage** method:

1. Call the **CreateMessage** method to begin the message composition. The sender email address should generally be the same address as the one used in the "From" header field in the message.
2. Call the **AddRecipient** method for each recipient of the message. These addresses are typically specified in the "To" and "Cc" header fields in the message. Additional addresses may also be be provided which are not specified in the email message itself. This is how one or more blind carbon copies of a message is delivered. Most servers have a limit on the total number of recipients that may be specified for a single message. This limit is usually around 100 addresses.
3. Call the **Write** method to write the contents of the message to the data stream. The application may also choose to use the **AppendMessage** method to write out a large amount of message data.
4. Call the **CloseMessage** method to close the message and submit it to the mail server for delivery.

For applications that do not need to dynamically compose the message and already have the message contents stored in a file or memory buffer, the **SendMessage** method is the preferred method of submitting a message for delivery.

See Also

AddRecipient Method, AppendMessage Method, CloseMessage Method, SendMessage Method,

Write Method

Disconnect Method

Terminate the connection with a server.

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

IsConnected Property, Connect Method, OnDisconnect Event

ExpandAddress Method

Expand the specified email address.

Syntax

object.**ExpandAddress**(*Address*, *ExpandedAddress*)

Parameters

Address

A string argument which specifies the address to expand.

ExpandedAddress

A string argument which will contain the list of expanded addresses when the method returns.

Return Value

If the method succeeds, it will return a value of true. If it was unable to expand the address, then the return value will be False. If the method fails, the **ResultString** property may provide additional information as to why the failure occurred.

Remarks

The **ExpandAddress** method requests that the server expand the specified email address. Typically this is used to expand aliases which refer to a mailing list, returning all of the members of that list. A server may not support this command, or may restrict its usage. An application should not depend on the ability to expand addresses.

See Also

VerifyAddress 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 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 smtpClient = CreateObject("SocketTools.SmtpClient.11")

nError = smtpClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize 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, open network connections will be closed and any handles allocated by the control will be released.

See Also

Cancel Method, Initialize Method, Uninitialize Method

SendMessage Method

Submit the specified message to the mail server for delivery.

Syntax

object.**SendMessage**(*Sender*, *Recipients*, *Message*, [*Options*])

Parameters

Sender

A string argument which specifies the email address of the person sending the message. This typically corresponds to the address in the From header of the message, but it is not required that they be the same.

Recipients

A string argument which specifies the email address of the person or persons to receive the message. Multiple addresses may be specified by separating each address with a comma. It should be noted that this protocol is only concerned with the delivery of a message and not its contents. Header fields in the message are not parsed to automatically determine the recipients. This argument should be a concatenation of all recipients, including carbon copies and blind carbon copies, with each address separated with a comma.

Message

A string argument contains the message to be delivered to the specified recipients. The message must be text and conform to the basic structure defined in RFC 822. There must be one or more headers separated by a blank line, followed by the body of the message. Each line of text must be terminated by a carriage return and linefeed character sequence. Note that more complex multipart MIME messages may also be used, but it is recommended that you use the Mail Message control to compose them.

Options

An optional argument that is reserved for future use. This argument should be omitted when calling this method.

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 **SendMessage** method enables an application to send a formatted email message using the current mail server. This provides a convenient one-step method of addressing and sending a message, and is designed to easily integrate with the Mail Message control.

This method will cause the current thread to block until the message transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

Example

The following example demonstrates how to use the **SendMessage** method, along with the Mail Message control, to compose and deliver a message:

```
' Compose a new message
nError = MailMessage1.ComposeMessage(strFrom, _
                                     strTo, _
```

```
                                        strCc, _
                                        strBcc, _
                                        strSubject, _
                                        strMessageText)

If nError > 0 Then
    MsgBox MailMessage1.LastErrorString, vbExclamation
    Exit Sub
End If

If MailMessage1.Recipients = 0 Then
    MsgBox "There are no recipients for this message"
    Exit Sub
End If

' Connect to the mail server
nError = SmtpClient1.Connect(strMailServer)
If nError > 0 Then
    MsgBox SmtpClient1.LastErrorString, vbExclamation
    Exit Sub
End If

' Deliver the message
nError = SmtpClient1.SendMessage(MailMessage1.Sender, MailMessage1.AllRecipients,
MailMessage1.Message)
If nError > 0 Then
    MsgBox SmtpClient1.LastErrorString, vbExclamation
    SmtpClient1.Disconnect
    Exit Sub
End If

' Disconnect
SmtpClient1.Disconnect
```

See Also

AddRecipient Method, Connect Method, CreateMessage Method, Disconnect 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

VerifyAddress Method

Verify the specified email address.

Syntax

object.**VerifyAddress**(*Address*, *VerifiedAddress*)

Parameters

Address

A string argument which specifies the address to verify.

VerifiedAddress

A string argument which will contain the verified address when the method returns. This parameter must be passed by reference.

Return Value

If the method succeeds, it will return a value of true. If it was unable to verify the address, then the return value will be False. If the method fails, the **ResultString** property may provide additional information as to why the failure occurred.

Remarks

The **VerifyAddress** method requests that the server verify the specified email address. Typically this is used to verify that a recipient address is valid, and return a fully qualified email address for that recipient. A server may not support this command, or may restrict its usage. An application should not depend on the ability to verify addresses.

See Also

ExpandAddress Method

Write Method

Write data to the server.

Syntax

object.**Write**(*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 string contains Unicode characters, it will be automatically converted to use standard UTF-8 encoding prior to being sent. If you wish to send the data without conversion, use a **Byte** array as the buffer instead of a **String** variable.

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 server, or -1 if an error was encountered.

Remarks

The **Write** method sends the data in ***buffer*** to the server. 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 blocking, the application will wait until the data can be sent. If the control is non-blocking and the write fails because it could not send all of the data to the server, the **OnWrite** event will be fired when the server can accept data again.

If the **Write** method is used to send the message contents to the server, the application must first call the **CreateMessage** method to specify the sender and the length of the message, followed by one or more calls to the **AddRecipient** method to specify each recipient of the message. When all of the message text has been submitted to the server, the application must call the **CloseMessage** method.

The message text is filtered by the **Write** method, and it will automatically normalize end-of-line character sequences to ensure the message meets the protocol requirements. The message itself must be in a standard RFC 822 or multi-part MIME message format, or the server may reject the message. Binary data, such as file attachments, should always be encoded. The **MailMessage** control can be used to compose and export a message in the correct format, which can then be submitted to the server.

It is recommended that most applications use the **SendMessage** method, which submits the message in a single method call.

See Also

AddRecipient Method, CloseMessage Method, CreateMessage Method, SendMessage Method

Simple Mail Transfer Protocol Control Events

Event	Description
OnCancel	This event is generated when a blocking operation is canceled
OnCommand	This event is generated when the server processes a command issued by the client
OnConnect	This event is generated when a connection is established
OnDisconnect	This event is generated when a connection is terminated
OnError	This event is generated when a control error occurs
OnProgress	This event is generated during data transfer
OnTimeout	This event is generated when a blocking operation times out

OnCancel Event

The **OnCancel** event is generated when a blocking operation is canceled.

Syntax

Sub *object*_**OnCancel** ([*Index* **As Integer**])

Remarks

This event is generated 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, OnTimeout Event

OnCommand Event

The **OnCommand** event is generated when the client sends a command to the server and receives a reply indicating the results of that command.

Syntax

Sub *object*_**OnCommand**([*Index* **As Integer**], **ByVal** *ResultCode* **As Variant**, **ByVal** *ResultString* **As Variant**)

Remarks

The **OnCommand** event is generated when the client receives a reply from the server after some action has been taken. The ***ResultCode*** argument contains the numeric result code returned by the server. The result codes returned from the server fall into one of the following categories:

Value	Description
100-199	Positive preliminary result. This indicates that the requested action is being initiated, and the client should expect another reply from the server before proceeding.
200-299	Positive completion result. This indicates that the server has successfully completed the requested action.
300-399	Positive intermediate result. This indicates that the requested action cannot complete until additional information is provided to the server.
400-499	Transient negative completion result. This indicates that the requested action did not take place, but the error condition is temporary and may be attempted again.
500-599	Permanent negative completion result. This indicates that the requested action did not take place.

The ***ResultString*** argument contains the descriptive string returned by the server which describes the result. The string contents may vary depending on the type of server.

See Also

ResultCode Property, ResultString Property, Command Method

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 server as a result of a **Connect** method call. This event is only triggered when the **Blocking** property is set to False.

See Also

Blocking Property, 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 server. This event is only triggered when the **Blocking** property is set to False.

When the **OnDisconnect** event fires, it is possible that there may still be buffered data available to read from the server. Before disconnecting from the server, the application should attempt to read any remaining data until the **Read** method returns a value of zero, or returns an error indicating that the operation would block.

See Also

Blocking Property, IsConnected Property, Connect Method, Disconnect Method, 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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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 during data transfer.

Syntax

Sub *object*_**OnProgress** ([*Index* **As Integer**], **ByVal** *MessageSize* **As Variant**, **ByVal** *MessageCopied* **As Variant**, **ByVal** *Percent* **As Variant**)

Remarks

The **OnProgress** event is generated during the transfer of data between the client and server, indicating the amount of data exchanged. For transfers of 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:

MessageSize

The size of the message being transferred in bytes.

MessageCopied

The number of bytes that have been transferred between the client and server.

Percent

The percentage of data that's been transferred, expressed as an integer value between 0 and 100, inclusive.

Note that this event is only generated when a message is delivered using the **SendMessage** method. If the client is writing the message data directly to the server using **Write** method then the application is responsible for calculating the completion percentage and updating any user interface controls.

See Also

SendMessage Method

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. This event is only triggered when the **Blocking** property is set to True.

See Also

Timeout Property, OnCancel Event

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 triggered when the **Blocking** property is set to False.

See Also

IsWritable Property, Write Method, OnConnect Event

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	CSWSKX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.SocketWrench.11
ClassID	B2880FA8-3F91-40C1-B8A1-D63FF4B9FF9B
Threading Model	Apartment
Help File	CSW11HLP.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 Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. 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

SocketWrench is a self-registering ActiveX control compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0, you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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	Enable or disable error handling by the container of the control
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 object

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 returned 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
Text sent and received using a string should be converted using the ANSI code page for the current locale. This is the default encoding type.	
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. It is not recommended that you use this code page unless you know that the remote host is sending text which includes OEM characters.	
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.	
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.	
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.	
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 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. 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. 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 LAN.

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 persistent broadband connections.

It is important to note that checking this property value may cause the 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.

Data Type

String

See Also

HostAddress Property, LocalAddress Property, PeerAddress 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

Options Property

Gets and sets the options that are used in establishing a connection.

Syntax

object.**Options** [= *value*]

Remarks

The **Options** property is an integer value which specifies one or more options. The value specified for this property will be used as the default options when connecting to the server. The property value is created by using a bitwise operator with one or more of the following values:

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.

&H10	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.
&H20	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.
&H800	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.
&H1000	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.
&H8000	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.
&H40000	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.

Several of these options correspond to Boolean properties which can be used to enable or disable specific functionality. For example, setting the **Secure** to True is the same as specifying the **swOptionSecure** option. We generally recommend setting individual properties when they are available, or explicitly specifying the required options when calling the **Connect** method.

Data Type

Integer (Int32)

See Also

Broadcast Property, InLine Property, NoDelay Property, Secure Property, SecureProtocol Property, Connect Method

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

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

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
Endif
```

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
Endif
On Error GoTo 0
```

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 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 socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable (or disable) the tracing of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named CSTRACE.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
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).

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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing 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
swTraceInfo	All function calls are written to the trace file. This is the default value.
swTraceError	Only those function calls which fail are recorded in the trace file.
swTraceWarning	Only those function calls which fail, or return values which indicate a warning, are recorded in the trace file.
swTraceHexDump	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.

Since socket function tracing is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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 the error WSAEWOULDBLOCK is returned, a warning is generated since the application simply needs to attempt to write the data at a later time.

Data Type

String

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 object.

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

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	
&H1000	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 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.
&H10	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.
&H20	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.
&H800	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.
&H1000	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.
&H8000	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.
&H40000	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

An 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.
&H10	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.
&H20	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.
&H800	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.
&H1000	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.
&H8000	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.
&H40000	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 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 is generated when a remote host connects to a listening socket
OnCancel	This event is generated when a blocking operation is canceled
OnConnect	This event is generated when a connection is established
OnDisconnect	This event is generated when a connection is terminated
OnError	This event is generated when a control error occurs
OnProgress	This event is generated as a data stream is being read or written
OnRead	This event is generated when data is available to be read
OnTimeout	This event is generated when a blocking operation times out
OnTimer	This event is generated when the control's preset timer interval expires
OnWrite	This event is generated 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 is generated 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 is generated 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 is generated 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 is generated 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

Secure Shell Protocol Control

Establish an interactive terminal session with an SSH server and execute remote commands.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	SshClientCtl.SshClient
File Name	CSTSHX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.SshClient.11
ClassID	46A9CAC5-33A2-4018-AA39-8CAB904A7294
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 4251

Overview

The Secure Shell (SSH) protocol ActiveX control is used to establish a secure connection with a server which provides a virtual terminal session for a user. Its functionality is similar to how character based consoles and serial terminals work, enabling a user to login to the server, execute commands and interact with applications running on the server. The control provides an interface for establishing the connection and handling the standard I/O functions needed by the program. It also includes methods that enable a program to easily scan the data stream for specific sequences of characters, making it very simple to write light-weight client interfaces to applications running on the server. This control can be combined with the Terminal Emulation control to provide complete terminal emulation services for a standard ANSI or DEC-VT220 terminal.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows

operating system.

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.

Secure Shell Protocol Control Properties

Property	Description
AutoResolve	Determines if host names and IP addresses are automatically resolved
Blocking	Gets and sets the blocking state of the control
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
Columns	Gets and sets the number of columns for the virtual terminal session
Command	Gets and sets the command that will be executed on the server
ExitCode	Return the exit code from the command executed on the server
Fingerprint	Returns a string that uniquely identifies the server
HashStrength	Return the length of the message digest that was selected
HostAddress	Gets and sets the IP address of the server
HostName	Gets and sets the name of the server
IsBlocked	Return if the control is blocked performing an operation
IsConnected	Determine if the control is connected to a server
IsInitialized	Determine if the control has been initialized
IsReadable	Return if data can be read from the server without blocking
IsWritable	Return if data can be sent to the server without blocking
KeepAlive	Gets and sets a value which determines if the client session should kept active
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error to occur
NewLine	Gets and sets the end-of-line character sequences sent to the server
Options	Gets and sets the options that are used in establishing a connection
Password	Gets and sets the password used to authenticate the client session
PrivateKey	Gets and sets the name of the private key file used to authenticate the client session
ProxyHost	Gets and sets the hostname or IP address for the proxy server
ProxyPassword	Gets and sets the password that will be used to authenticate the proxy connection
ProxyPort	Gets and sets the port number for the proxy server
ProxyType	Gets and sets the type of proxy server the connection will be established through
ProxyUser	Gets and sets the user name that will be used to authenticate the proxy connection
RemotePort	Gets and sets the port number for a remote connection
Rows	Gets and sets the number of rows for the virtual terminal session
Secure	Set or return if a connection to the server is secure
SecureCipher	Return the encryption algorithm used to establish the secure connection with the server
SecureHash	Return the message digest selected when establishing the secure connection with the server

SecureKeyExchange	Return the key exchange algorithm used to establish the secure connection with the server
SecureProtocol	Gets and sets the security protocol used to establish the secure connection with the server
Terminal	Gets and sets the terminal type used by the control
ThrowError	Enable or disable error handling by the container of the control
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
UserName	Gets and sets the current user name used to authenticate the client session
Version	Return the current version of the object

AutoResolve Property

Determines if host names and IP 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 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 **AutoResolve** to False.

Data Type

Boolean

See Also

HostAddress Property, HostName Property

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 read data when none has been written, an error is generated. Events such as **OnConnect**, **OnDisconnect**, **OnRead** and **OnWrite** are only fired if the connection is non-blocking.

Data Type

Boolean

See Also

IsBlocked Property, IsReadable Property, IsWritable 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
Text sent and received using a string should be converted using the ANSI code page for the current locale. This is the default encoding type.	
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. It is not recommended that you use this code page unless you know that the remote host is sending text which includes OEM characters.	
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.	
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.	
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.	
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 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

Columns Property

Gets and sets the number of columns for the virtual terminal session.

Syntax

object.**Columns** [= *columns*]

Remarks

The **Columns** property returns the number of character columns for the virtual display. Setting this property prior to calling the **Connect** method requests that the server create a pseudoterminal with the specified number of columns. This property value is only meaningful for interactive terminal sessions, and is not used when executing a command on the server.

The default value for this property is 80.

Data Type

Integer (Int32)

See Also

Rows Property, Terminal Property, Connect Method

Command Property

Gets and sets the command that will be executed on the server.

Syntax

object.**Command** [= *command*]

Remarks

The **Command** property is used to specify a command and its arguments that should be executed on the server. The output of the command will be returned to the application and can be read using the **Read** or **ReadLine** method. If no command is specified, then the control will establish an interactive terminal session instead.

The command and its arguments must follow the conventions used by the SSH server, and the command will execute in the context of the authenticated user. The **ExitCode** property can be used to obtain the numerical exit code of the remote program, if one is available.

Data Type

String

See Also

ExitCode Property, Connect Method, Execute Method, Read Method, ReadLine Method

ExitCode Property

Return the exit code from the command executed on the server.

Syntax

object.**ExitCode**

Remarks

The **ExitCode** property returns the numeric exit code for the command that was previously executed on the server. This property value is only meaningful after the command has completed and the connection closed by calling the **Disconnect** method. In most cases, an exit code value of zero indicates success, while any other value indicates an error condition.

Note that the actual value is application dependent and is only meaningful in the context of that particular program. A program may choose to use exit codes in a non-standard way, such as having certain non-zero values indicate success.

The **Reset** method will reset the exit code back to its default value of zero.

Data Type

Integer (Int32)

See Also

Command Property, Connect Method

Fingerprint Property

Returns a string that uniquely identifies the server.

Syntax

object.**Fingerprint**

Remarks

The **Fingerprint** property returns a string that consists of a series of hexadecimal values separated by colons. The value is unique to the server, and is an MD5 hash of the RSA host key. An application can use this value to determine if a connection has been established with the server previously by storing the server's host name, IP address and fingerprint in a file, registry key or a database.

Data Type

String

See Also

Connect 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 server.

Syntax

object.**HostAddress** [= *ipaddress*]

Remarks

The **HostAddress** property can be used to set the IP address for a server 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, HostName Property

HostName Property

Gets and sets the name of the server.

Syntax

object.**HostName** [= *hostname*]

Remarks

The **HostName** property should be set to the name of the server 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

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.

Note that this property will return True if there is ***any*** blocking operation being performed by the application, regardless if the specified control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

Blocking Property, LastError Property

IsConnected Property

Determine if the control is connected to a server.

Syntax

object.**IsConnected**

Remarks

The **IsConnected** read-only property is set to a value of true if the control is connected with a server, otherwise the property has a value of false.

Data Type

Boolean

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

IsReadable Property

Return if data can be read from the server without blocking.

Syntax

object.**IsReadable**

Remarks

The **IsReadable** property returns True if data can be read from the server without blocking. For non-blocking connections, this property can be checked before the application attempts to read the data, preventing an error.

Data Type

Boolean

See Also

IsConnected Property, Read Method, OnRead Event

IsWritable Property

Return if data can be sent to the server without blocking.

Syntax

object.**IsWritable**

Remarks

The **IsWritable** property returns True if data can be written without blocking. For non-blocking connections, this property can be checked before the application attempts to send data to the server, 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 send data without an error. The next operation may result in an **stErrorOperationWouldBlock** or **stErrorOperationInProgress** error. The application must treat these errors as recoverable, and should be prepared to retry operations that result in them.

Data Type

Boolean

See Also

IsReadable Property, SendKey Method, Write Method, OnWrite Event

KeepAlive Property

Gets and sets a value which determines if the client session should kept active.

Syntax

object.**KeepAlive** [= { True | False }]

Remarks

Setting the **KeepAlive** property to a value of true indicates that the client wishes to maintain a long-duration session with the server. It is only necessary to set this property to a value of true if the client session is interactive and the connection must be held open for more than two hours.

ta Type

Boolean

See Also

Connect 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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

NewLine Property

Gets and sets the end-of-line character sequences sent to the server.

Syntax

object.**NewLine** [= *value*]

Remarks

The **NewLine** property is an integer value that specifies how newlines are sent to the server. It may be one of the following values:

Value	Description
sshNewLineDefault	There are no changes to how data is sent to the server. Any carriage return or linefeed characters that are sent using the **Write** method will be sent as-is. The **WriteLine** method will terminate each line of text with a carriage return and linefeed (CRLF) sequence. This is the default line mode that is set when a new connection is established.
sshNewLineCR	A carriage return is used as the end-of-line character. Any data sent using the **Write** method that contains only a linefeed (LF) character or a carriage return and linefeed (CRLF) sequence to indicate the end-of-line will be replaced by a carriage return (CR) character. The **WriteLine** method will terminate each line of text with a single carriage return character.
sshNewLineLF	A linefeed is used as the end-of-line character. Any data sent using the **Write** method that contains only a carriage return (CR) character or a carriage return an linefeed (CRLF) sequence to indicate the end-of-line will be replaced by a linefeed (LF) character. The **WriteLine** method will terminate each line of text with a single linefeed character.
sshNewLineCRLF	A carriage return and linefeed (CRLF) character sequence is used to indicate the end-of-line. Any data sent using the **Write** method that contains only a carriage return (CR) or linefeed (LF) will be replaced by a carriage return and linefeed. The **WriteLine** method will terminate each line of text with a carriage return and linefeed sequence.

When a connection is initially established with the server, it determines what characters are used to indicate the end-of-line and how they are displayed. On UNIX based systems, this is controlled by the settings for the pseudo-terminal that is allocated for the client session, and can be changed using the **stty** command. In most cases, the client line mode can be left at the default. However, in some cases you may need to change the line mode, particularly if you intend to send data from a Windows text file or copied from the clipboard.

Windows uses a carriage return and linefeed (CRLF) sequence to indicate the end-of-line and a UNIX based server may interpret that as multiple newlines. To prevent this, set the **NewLine** property to **sshNewLineCR** and the CRLF sequence in the text will be replaced by a single carriage return.

Data Type

Integer (Int32)

See Also

Options Property

Gets and sets the options that are used in establishing a connection.

Syntax

object.**Options** [= *value*]

Remarks

The **Options** property is an integer value which specifies one or more options. The value specified for this property will be used as the default options when connecting to the server. The property value is created by using a bitwise operator with one or more of the following values:

Value	Description
sshOptionNone	No options specified. A standard terminal session will be established with the default terminal type.
sshOptionKeepAlive	This option specifies the library should attempt to maintain an idle client session for long periods of time. This option is only necessary if you expect that the connection will be held open for more than two hours. This option is the same as setting the **KeepAlive** property to a value of true.
sshOptionNoPTY	This option specifies that a pseudoterminal (PTY) should not be created for the client session. This option is automatically set if the **Command** property specifies a command to be executed on the server.
sshOptionNoShell	This option specifies that a command shell should not be used when executing a command on the server.
sshOptionNoAuthRSA	This option specifies

	that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server.	
sshOptionNoPwdNul	This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server.	
sshOptionNoRekey	This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour.	
sshOptionCompatSID	This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.	
sshOptionCompatHMAC	This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.	
&H40000	sshOptionPreferIPv6	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.

Data Type

Integer (Int32)

See Also

Secure Property, Connect Method

Password Property

Gets and sets the password for the current user.

Syntax

object.**Password** [= *password*]

Remarks

The **Password** property specifies the password used to authenticate the user. The **UserName** and **Password** properties are used together to provide credentials to the server that will authenticate the client session. This property is used as the default value for the **Connect** method if no password is specified as an argument.

If you are connecting to a server that requires public/private key authentication, your application should set the **PrivateKey** property to the path of the user's private key. In this case, the **Password** property can be used to specify the password required to access the private key. If the private key was created without a password, this property should be set to an empty string.

Data Type

String

See Also

PrivateKey Property, UserName Property, Connect Method

PrivateKey Property

Gets and sets the private key file used for SSH authentication.

Syntax

object.**PrivateKey** [= *filename*]

Remarks

The **PrivateKey** property specifies the path to the private key file used to authenticate the user. The private key is used in combination with the value of the **UserName** property to provide credentials to an SSH server. If public/private key authentication is not required by the server, this property is should be set to an empty string.

The **PrivateKey** property value can use environment variables enclosed in percent symbols, and the path to the private key file will be normalized. It is recommended you always use an absolute path to the private key file. If you do not include a path, it will use the current working directory for the process. This can produce inconsistent results because the current working directory for a process is a global value and it can be changed at any time.

The private key file name must resolve to a text file which can be read by the current process. If the file does not exist, or it does not specify a PEM formatted file which contains an RSA or OpenSSH private key, the connection will fail and the last error code will be set to `stErrorInvalidPrivateKey`.

Data Type

String

Example

```
SshClient1.UserName = "username"
SshClient1.PrivateKey = "%USERPROFILE%\\.ssh\\id_rsa.pem"

nError = SshClient1.Connect("name.server.tld", 22)
If nError > 0 Then
    MsgBox SshClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

In this example, the path to the private key file will be expanded to an absolute path because the USERPROFILE environment variable defines the home directory for the current user.

See Also

Password Property, UserName Property, Connect Method

ProxyHost Property

Gets and sets the host name of the proxy server.

Syntax

object.**ProxyHost** [= *hostname*]

Remarks

The **ProxyHost** property should be set to the name of the proxy server that you want to connect through. This property may be set to either a fully qualified domain name, or an IP address. This property is only used if the **ProxyType** property is set to a non-zero value.

Data Type

String

See Also

ProxyPassword Property, ProxyPort Property, ProxyType Property, ProxyUser Property, Connect Method

ProxyPassword Property

Gets and sets the proxy server password for the current user.

Syntax

object.**ProxyPassword** [= *password*]

Remarks

The **ProxyPassword** property specifies the password used to authenticate the user to the proxy server. If a password is not required by the server, this property is ignored.

Data Type

String

See Also

ProxyHost Property, ProxyPort Property, ProxyType Property, ProxyUser Property, Connect Method

ProxyPort Property

Gets and sets the port number for the proxy server.

Syntax

object.**ProxyPort** [= *portnumber*]

Remarks

The **ProxyPort** property is used to set the port number that the control will use to establish a connection with the proxy server. A value of zero specifies that the client will connect to the proxy server using the default port for the selected proxy type. If the client is connecting through an HTTP proxy, the connection will be established on port 80 by default. If the client is connecting through a Telnet proxy, the connection will be established on port 23 by default.

Data Type

Integer (Int32)

See Also

ProxyHost Property, ProxyPassword Property, ProxyType Property, ProxyUser Property, Connect Method

ProxyType Property

Gets and sets the current proxy server type.

Syntax

object.**ProxyType** = [*proxytype*]

Remarks

The **ProxyType** property specifies the type of proxy server that the client is connecting to. The supported proxy server types are as follows:

Value	Description
sshProxyNone	A direct connection will be established with the server.
sshProxyHttp	Establish a connection through a proxy server using the Hypertext Transfer Protocol.
sshProxyTelnet	Establish a connection through a proxy server using the Telnet protocol.

If the **sshProxyNone** proxy type is specified, then a direct connection is established to the server and the proxy-related properties are ignored. If a port number for the proxy server is not explicitly specified, then the default port number for the proxy server type will be used. If the client is connecting through an HTTP proxy, the connection will be established on port 80 by default. If the client is connecting through a Telnet proxy, the connection will be established on port 23 by default.

Data Type

Integer (Int32)

See Also

ProxyHost Property, ProxyPassword Property, ProxyPort Property, ProxyUser Property, Secure Property, Connect Method

ProxyUser Property

Gets and sets the current proxy user name.

Syntax

object.**ProxyUser** [= *username*]

Remarks

The **ProxyUser** property specifies the user that is logging in to the proxy server. If the proxy server does not require user authentication, then this property is ignored.

Data Type

String

See Also

ProxyHost Property, ProxyPassword Property, ProxyPort Property, ProxyType Property, Connect Method

RemotePort Property

Gets and sets the port number for a remote connection.

Syntax

object.**RemotePort** [= *portnumber*]

Remarks

The **RemotePort** property is used to set the port number that the control will use to establish a connection with the server.

Data Type

Integer (Int32)

See Also

HostAddress Property, HostName Property

Rows Property

Gets and sets the number of rows for the virtual terminal session.

Syntax

object.**Rows** [= *rows*]

Remarks

The **Rows** property returns the number of character rows for the virtual display. Setting this property prior to calling the **Connect** method requests that the server create a pseudoterminal with the specified number of rows. This property value is only meaningful for interactive terminal sessions, and is not used when executing a command on the server.

The default value for this property is 24.

Data Type

Integer (Int32)

See Also

Columns Property, Terminal Property, Connect Method

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 true, and it is included only for compatibility with the other SocketTools components. Because all SSH connections must be secure, attempting to set this property to a value of false will result in an error.

Data Type

Boolean

See Also

Connect Method, Initialize 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
stCipherNone	No cipher has been selected. This is not a secure connection with the server.
stCipherRC2	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.
stCipherRC4	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.
stCipherRC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
stCipherDES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher using 56-bit keys.
stCipherDES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively using a 168-bit key length.
stCipherDESX	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.
stCipherAES	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.
stCipherSkipjack	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
stCipherBlowfish	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
stHashNone	No message digest algorithm has been selected. This is not a secure connection with the server.
stHashMD5	The MD5 algorithm was selected. This algorithm takes a message of arbitrary length and produces a 128-bit message digest.
stHashSHA	The SHA algorithm was selected. This algorithm produces a 160-bit message digest.

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
stKeyExchangeNone	No key exchange algorithm has been selected. This is not a secure connection with the server.
stKeyExchangeRSA	The RSA public key exchange algorithm has been selected.
stKeyExchangeKEA	The KEA public key exchange algorithm has been selected. This is an improved version of the Diffie-Hellman public key algorithm.
stKeyExchangeDH	The Diffie-Hellman public key exchange algorithm has been selected.
stKeyExchangeECDH	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 either SSH-1 or SSH-2 to establish the connection, with the appropriate protocol automatically selected based on the capabilities of the 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. Because all connections to an SSH server are secure, this value indicates that a connection has not been established.
stProtocolSSH1	The Secure Shell 1.0 protocol should be used. This version of the protocol has been deprecated and is no longer widely used. It is not recommended that this version of the protocol be used to establish a connection.
stProtocolSSH2	The Secure Shell 2.0 protocol should be used. This is the most commonly used version of the protocol. It is recommended that this version of the protocol be used unless the server explicitly requires the client to use an earlier version.

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 before calling the **Connect** method.

Data Type

Integer (Int32)

See Also

CipherStrength Property, HashStrength Property, Secure Property, SecureCipher Property, SecureHash Property, SecureKeyExchange Property

Terminal Property

Gets and sets the terminal type used by the control.

Syntax

object.**Terminal** [= *termtype*]

Remarks

The **Terminal** property specifies the terminal type of the server for display purposes. On UNIX based systems, the terminal name corresponds to a termcap or terminfo entry as set in the TERM environment variable. On Windows based systems which implement the ssh service, this property may be ignored and the server will assume that the client is capable of displaying ANSI escape sequences. On VMS systems, the terminal name should correspond to the terminal type used with the SET TERMINAL/DEVICE command.

If this property is set to an empty string and no terminal type is specified when the **Connect** method is called, a default terminal type named "unknown" will be used. On most UNIX and VMS systems this defines a terminal which is not capable of cursor positioning using control or escape sequences. This terminal type may not be recognized and an error may be displayed when the user logs in indicating that the terminal type is invalid.

Refer to the documentation for the server system to determine what terminal type names are available to you. Remember that on UNIX systems, the terminal type is case-sensitive. Some of the more common terminal types are:

Terminal Type	Description
ansi	This terminal type is usually available on UNIX based servers. This specifies that the client is capable of displaying standard ANSI escape sequences for cursor control.
dumb	This terminal type typically specifies a terminal display which does not support control or escape sequences for cursor positioning. If you do not want escape sequences embedded in the data stream and the server returns an error if the terminal type is not specified, try using this terminal type.
pcansi	This terminal type is usually available on UNIX based servers. This specifies that the client is a using a PC terminal emulator that supports basic ANSI escape sequences for cursor control. This may also enable escape sequences which can set the display colors.
vt100	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT100. This specifies that the client is capable of emulating a DEC VT100 terminal. The VT100 supports many of the same cursor control sequences as an ANSI terminal.
vt220	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT220. This specifies that the client is capable of emulating a DEC VT220 terminal, which is a later version of the VT100.
vt320	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT320.

	This specifies that the client is capable of emulating a DEC VT320 terminal, which is similar to the VT100 and VT220 and provides advanced features such as the ability to set display colors.
xterm	This terminal type is may be available on UNIX based servers which have X Windows installed. This specifies that the client is a using the X Windows xterm emulator which supports standard ANSI escape sequences for cursor control.

Data Type

String

See Also

Columns Property, Rows Property

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
SshClient1.ThrowError = False
nError = SshClient1.Connect(strHostName)

If nError > 0 Then
    MsgBox SshClient1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

SshClient1.ThrowError = True
SshClient1.Connect strHostName

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error.

Note that the **Timeout** property also determines the amount of time the control will spend attempting to connect to a server. If a connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
sshTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
sshTraceError	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.
sshTraceWarning	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.
sshTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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

UserName Property

Gets and sets the current user name.

Syntax

object.**UserName** [= *username*]

Remarks

The **UserName** property identifies the user that is connecting to the server and is required for authentication purposes. This property is used as the default value for the **Connect** method if no username is specified as an argument.

Data Type

String

See Also

Password Property, PrivateKey Property, Connect Method

Version Property

Return the current version of the object.

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

Secure Shell Protocol Control Methods

Method	Description
Break	Sends a break signal to the server
Cancel	Cancels the current blocking network operation
Connect	Establish a connection with a server
Control	Send a control message to the server
Disconnect	Terminate the connection with the server
Execute	Execute a command on the server and return the output
Initialize	Initialize the control and validate the runtime license key
Peek	Read data returned by the server, but do not remove it from the receive buffer
Read	Return data read from the server
ReadLine	Read a line of text from the server
Reset	Reset the internal state of the control
Search	Search for a specific character sequence in the data stream
SendKey	Send a key code to the server
Uninitialize	Uninitialize the control and release any system resources that were allocated
Write	Write data to the server
WriteLine	Write a line of text to the server

Break Method

Sends a break signal to the server.

Syntax

object.**Break**

Parameters

None.

Return Value

A value of zero is returned if the break signal was sent successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Break** method a control message to the server which simulates a break signal on a physical terminal. This is used by some operating systems as an instruction to enter a privileged configuration mode. Note that this is not the same as sending an interrupt character such as Ctrl+C to the server. This control code is ignored for SSH 1.0 sessions.

See Also

Cancel Method, Control 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

Break Method, Control Method

Connect Method

Establish a connection with a server.

Syntax

object.**Connect**([*RemoteHost*], [*RemotePort*], [*UserName*], [*Password*], [*Timeout*], [*Options*])

Parameters

RemoteHost

A string which specifies the host name or IP address of the server. If this argument is not specified, it defaults to the value of the **HostAddress** property if it is defined. Otherwise, it defaults to the value of the **HostName** property.

RemotePort

A number which specifies the port to connect to on the server. If this argument is not specified, it defaults to the value of the **RemotePort** property. A value of zero indicates that the default port number for this service should be used to establish the connection.

UserName

A string which specifies the user name which will be used to authenticate the client session. This value must specify a valid user name and cannot be an empty string. If this argument is not specified, it defaults to the value of **UserName** property.

Password

A string which specifies the password which will be used to authenticate the client session. If the user does not have a password, this value can be an empty string. If this argument is not specified, it defaults to the value of the **Password** property. If the server requires public/private key authentication, set the **PrivateKey** property to the name of the file containing the private key prior to calling this method.

Timeout

The number of seconds that the client will wait for a response before failing the operation. If this argument is not specified, the value of the **Timeout** property will be used as the default.

Options

A numeric value which specifies one or more options. If this argument is omitted or a value of zero is specified, a default connection will be established. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description
sshOptionNone	No options specified. A standard terminal session will be established with the default terminal type.
sshOptionKeepAlive	This option specifies the library should attempt to maintain an idle client session for long periods of time. This option is only necessary if you expect that the connection will

	be held open for more than two hours. This option is the same as setting the **KeepAlive** property to a value of true.	
sshOptionNoPTY	This option specifies that a pseudoterminal (PTY) should not be created for the client session. This option is automatically set if the **Command** property specifies a command to be executed on the server.	
sshOptionNoShell	This option specifies that a command shell should not be used when executing a command on the server.	
&H40000	sshOptionPreferIPv6	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.

See Also

HostAddress Property, HostName Property, Options Property, Password Property, PrivateKey Property, RemotePort Property, UserName Property, Disconnect Method, OnConnect Event

Control Method

Send a control message to the server.

Syntax

object.**Control**(*ControlCode*)

Parameters

ControlCode

A numeric control code which specifies the control message which should be sent to the server. This may be one of the following values:

Value	Description
sshControlBreak	Sends a control message to the server which simulates a break signal on a physical terminal. This is used by some operating systems as an instruction to enter a privileged configuration mode. Note that this is not the same as sending an interrupt character such as Ctrl+C to the server. This control code is ignored for SSH 1.0 sessions. This is the same as calling the **Break** method.
sshControlNoop	Sends a control message to the server, but it does not perform any operation. This is typically used by clients to prevent the server from automatically closing a session that has been idle for a long period of time.
sshControlEof	Sends a control message to the server indicating that the client has finished sending data. Note that this option is normally not used with interactive terminal sessions, and should only be used when required by the server.
sshControlPing	Sends a control message to the server which is used to test whether or not the server is responsive to the client. This is typically used by clients to attempt to detect if the connection to the server is still active.
sshControlRekey	Sends a control message to the server requesting that the key exchange be performed again. This control code is ignored for SSH 1.0 sessions.

Return Value

A value of zero is returned if the control code was sent successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Control** method enables an application to send control messages to the server, which can cause it to take specific actions such as simulate a terminal break or request that the key exchange be performed again. Some control messages are not supported by the SSH 1.0 protocol, in which case the control message is ignored.

See Also

Break Method, Cancel Method

Disconnect Method

Terminate the connection with a server.

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

IsConnected Property, ExitCode Property, Connect Method, OnDisconnect Event

Execute Method

Execute a command on the server and return the output.

Syntax

object.**Execute**([*RemoteHost*], [*RemotePort*], [*UserName*], [*Password*], [*Command*], [*Timeout*], [*Options*])

Parameters

RemoteHost

A string which specifies the host name or IP address of the server. If this argument is not specified, it defaults to the value of the **HostAddress** property if it is defined. Otherwise, it defaults to the value of the **HostName** property.

RemotePort

A number which specifies the port to connect to on the server. If this argument is not specified, it defaults to the value of the **RemotePort** property. A value of zero indicates that the default port number for this service should be used to establish the connection.

UserName

A string which specifies the user name which will be used to authenticate the client session. If this argument is not specified, it defaults to the value of **UserName** property.

Password

A string which specifies the password which will be used to authenticate the client session. If this argument is not specified, it defaults to the value of the **Password** property.

Command

A string which specifies the command that will be executed on the server. If this argument is not specified, it defaults to the value of the **Command** property.

Timeout

The number of seconds that the client will wait for a response before failing the operation. If this argument is not specified, the value of the **Timeout** property will be used as the default.

Options

A numeric value which specifies one or more options. If this argument is omitted or a value of zero is specified, a default connection will be established. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description
sshOptionNone	No options specified. A standard terminal session will be established with the default terminal type.
sshOptionKeepAlive	This option specifies the library should attempt to maintain an idle client session for long periods of time. This option is only necessary if you expect that the connection will be held open for more than two hours. This option is the same as setting the **KeepAlive** property to a value of true.
sshOptionNoPTY	This option specifies that a pseudoterminal (PTY) should not be created for the client session. This option is automatically set if the **Command** property specifies a command to be executed

	on the server.
sshOptionNoShell	This option specifies that a command shell should not be used when executing a command on the server.
sshOptionNoAuthRSA	This option specifies that RSA authentication should not be used with SSH-1 connections. This option is ignored with SSH-2 connections and should only be specified if required by the server.
sshOptionNoPwdNul	This option specifies the user password cannot be terminated with a null character. This option is ignored with SSH-2 connections and should only be specified if required by the server.
sshOptionNoRekey	This option specifies the client should never attempt a repeat key exchange with the server. Some SSH servers do not support rekeying the session, and this can cause the client to become non-responsive or abort the connection after being connected for an hour.
sshOptionCompatSID	This compatibility option changes how the session ID is handled during public key authentication with older SSH servers. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.
sshOptionCompatHMAC	This compatibility option changes how the HMAC authentication codes are generated. This option should only be specified when connecting to servers that use OpenSSH 2.2.0 or earlier versions.

Return Value

A string that contains the output of the command that was executed on the server. To get the exit code returned by the program, check the value of the **ExitCode** property. If an empty string is returned, this indicates that there was either no data available, or an error has occurred and the **LastError** property will return a non-zero value.

Remarks

This method establishes a network connection with a server and executes the specified command. The output from the command is returned as a string. This method should not be used if the connection to the server must be established through a proxy server. If the connection must be made through a proxy server, then you should set the **Command** property to the specify the command to execute, call the **Connect** method to establish the connection, and then use either the **Read** or **ReadLine** methods to read the output.

When the command output is being read from the server, this method will automatically convert the data to match the end-of-line convention used on the Windows platform. This is useful when executing a command on a UNIX based system where the end-of-line is indicated by a single linefeed, while on Windows it is a carriage-return and linefeed pair. If the output contains embedded nulls or escape sequences, then this conversion will not be performed.

Example

The following example demonstrates how to use the Execute method and check for an error condition:

```
Dim strOutput As String

SshClient1.HostName = strServerName
SshClient1.UserName = strUserName
SshClient1.Password = strPassword
SshClient1.Command = "/bin/ls -l"

strOutput = SshClient1.Execute()

If Len(strOutput) = 0 Then
    If SshClient1.LastError > 0 Then
        MsgBox SshClient1.LastErrorString, vbExclamation
        Exit Sub
    End If
End If
```

See Also

Command Property, ExitCode Property, LastError Property, Connect 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 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 sshClient = CreateObject("SocketTools.SshClient.11")

nError = sshClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize Method

Peek Method

Read data returned by the server, but do not remove it from the receive 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

The number of bytes actually read from the server is returned by this method. If there is no data available to be read, a value of zero is returned. If an error occurs, a value of -1 is returned.

Remarks

The **Peek** method can be used to examine the data that is available to be read from the internal receive buffer. If there is no data in the receive buffer at that time, a value of zero is returned. It should be noted that this differs from the **Read** method, where a return value of zero indicates that there is no more data available to be read and the connection has been closed. The **Peek** method will never cause the client to block, and so may be safely used with asynchronous connections. Note that it is possible for the returned data to contain embedded null characters.

See Also

IsConnected Property, IsReadable Property, Read Method, ReadLine Method, Search Method, Write Method, OnRead Event, OnWrite Event

Read Method

Return data read from the server.

Syntax

object.**Read**(*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.

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 server 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 server, 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 stop until data is returned by the server or the connection is closed. Note that it is possible for the returned data to contain embedded null characters.

See Also

CodePage Property, IsConnected Property, IsReadable Property, ReadLine Method, Search Method, Write Method, OnRead Event

ReadLine Method

Read up to a line of data from the server and returns it in a string buffer.

Syntax

object.**ReadLine**(*Buffer*, [*Length*])

Parameters

Buffer

A string that the data will be stored in when the method returns. This parameter must be passed by reference.

Length

An optional parameter that specifies the maximum number of bytes to read. If this argument is omitted, then the control will return up to 4096 characters in the string. If the application expects that a single line of text will exceed this value, then it must be explicitly specified.

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 server 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 server 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 network operation is attempted while **ReadLine** is blocked waiting for data from the server, an error will occur. It is recommended that this method only be used with blocking 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, Timeout Property, Read Method, Write Method, WriteLine 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, open network connections will be closed and any handles allocated by the control will be released.

See Also

Cancel Method, Initialize Method, Uninitialize Method

Search Method

Search for a specific character sequence in the data stream.

Syntax

object.**Search**(*String*, [*Buffer*], [*Length*], [*Options*])

Parameters

String

A string value which specifies the sequence of characters to search for in the data stream. When the control encounters this sequence, the method will return.

Buffer

An optional string or byte array buffer that will contain the output sent by the server, up to and including the search string character sequence. If this argument is omitted, the control will still search for the character sequence but any output sent by the server will be discarded.

Length

An optional integer value which specifies the maximum number of bytes of data to store in the buffer. If this argument is omitted, no limit will be placed on the amount of output buffered by the control.

Options

An optional integer value which is reserved for future use. This argument should be omitted.

Return Value

This method returns a Boolean value. A return value of true indicates that the search string was found in the data stream. A return value of False indicates that the search string was not found in the amount of time specified by the **Timeout** property or that the server closed the connection.

Remarks

The **Search** method searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server, such as executing a command and processing the output. The function collects the output from the server and stores it in a buffer provided by the caller. When the function returns, the buffer will contain everything sent by the server up to and including the search string.

See Also

IsReadable Property, Timeout Property, Connect Method, Read Method, ReadLine Method

SendKey Method

Send a key code to the server.

Syntax

object.**SendKey**(*Key*)

Parameters

Key

A value which specifies the key code to send to the server. This may be a single byte, in which case it is sent to the server as-is. If a numeric value is specified, then this is considered to be an ASCII character value and it is sent to the server as a single byte. The value must be between 1 and 255. If the key code value is 0, then the method returns without sending any data. If the value is greater than 255, an error will be raised. If the *Key* argument is a string, then the method will send that string to the server. An empty string is ignored and the method will return without sending any data. An error will be returned if the string is longer than 128 bytes.

Return Value

This method will return a value of true if the key code was successfully sent to the server. If the key cannot be sent, the method will return False and the **LastError** property will contain the error code that indicates the reason for the failure. This method will also return False if the key code value is zero or an empty string is passed by the caller.

Remarks

The **SendKey** method sends a key code to the server. This method is useful if the application needs to send a single character to the server, as opposed to using the **Write** method which should be used for sending large amounts of data.

The strings sent by the **SendKey** method are typically short escape sequences which are generated by a terminal emulator when the user presses a special key, such as a function key. For example, a DEC VT100 terminal sends the escape sequence <ESC>[M when the user presses the F1 function key. To simulate this, those three bytes could be passed as the *Key* value.

Example

The following example demonstrates how to use the **SendKey** method in conjunction with the **KeyMapped** and **KeyPress** events in the Terminal Emulator control:

```
Private Sub Terminal1_KeyMapped(KeyIndex As Integer, Shift As Integer, KeyString As
String)
    SshClient1.SendKey KeyString
End Sub

Private Sub Terminal1_KeyPress(KeyAscii As Integer)
    SshClient1.SendKey KeyAscii
End Sub
```

See Also

IsWritable Property, Write Method, OnWrite 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 server.

Syntax

object.**Write**(*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 bytes actually written to the server, or -1 if an error was encountered.

Remarks

The **Write** method sends the data in ***buffer*** to the server. 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 blocking, the application will wait until the data can be sent. If the control is non-blocking and the write fails because it could not send all of the data to the server, the **OnWrite** event will be fired when the server can accept data again.

See Also

CodePage Property, IsConnected Property, IsWritable Property, Timeout Property, Read Method, ReadLine Method, SendKey Method, WriteLine Method, OnWrite Event

WriteLine Method

Send a line of text to the server, terminated by a carriage-return and linefeed.

Syntax

object.**WriteLine**([*Buffer*])

Parameters

Buffer

A string which contains the data that will be sent to the server. 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. 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 has been written to the server. If an error occurs, the method will return False.

Remarks

The **WriteLine** method writes a line of text to the server 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.

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 server 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 server, 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

CodePage Property, IsWritable Property, Timeout Property, Read Method, ReadLine Method, Write Method

Secure Shell Protocol Control Events

Event	Description
OnCancel	This event is generated when a blocking operation is canceled
OnConnect	This event is generated when a connection is established
OnDisconnect	This event is generated when a connection is terminated
OnError	This event is generated when a control error occurs
OnRead	This event is generated when data is available to be read
OnTimeout	This event is generated when a blocking operation times out
OnWrite	This event is generated when data can be written to the server

OnCancel Event

The **OnCancel** event is generated when a blocking operation is canceled.

Syntax

Sub *object*_**OnCancel** ([*Index* **As Integer**])

Remarks

This event is generated 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, OnTimeout 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 server as a result of a **Connect** method call. This event is only triggered when the **Blocking** property is set to False.

See Also

Blocking Property, Connect Method, OnDisconnect Event, OnWrite 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. This event is only triggered when the **Blocking** property is set to False.

When the **OnDisconnect** event fires, it is possible that there may still be buffered data available to read from the server. Before disconnecting from the server, the application should attempt to read any remaining data until the **Read** method returns a value of zero, or returns an error indicating that the operation would block.

See Also

Blocking Property, IsConnected Property, IsReadable Property, Connect Method, Disconnect Method, Read Method, 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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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

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 triggered when the **Blocking** property is set to False.

See Also

IsReadable Property, Read Method, ReadLine Method, Write Method, WriteLine 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. This event is only triggered when the **Blocking** property is set to True.

See Also

Timeout Property, OnCancel Event

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 triggered when the **Blocking** property is set to False.

See Also

IsWritable Property, Read Method, ReadLine Method, SendKey Method, Write Method, WriteLine Method, OnConnect Event, OnRead Event

Telnet Protocol Control

Establish an interactive terminal session with a server.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	TelnetClientCtl.TelnetClient
File Name	CSTNTX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.TelnetClient.11
ClassID	20552896-1108-4EC5-95B3-19741C1CF8BC
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 854

Overview

The Telnet protocol is used to establish a connection with a server which provides a virtual terminal session for a user. Its functionality is similar to how character based consoles and serial terminals work, enabling a user to login to the server, execute commands and interact with applications running on the server. The control provides an interface for establishing the connection, negotiating certain options (such as whether characters will be echoed back to the client) and handling the standard I/O functions needed by the program.

The control also provides methods that enable a program to easily scan the data stream for specific sequences of characters, making it very simple to write light-weight client interfaces to applications running on the server. This control can be combined with the SocketTools Terminal control to provide complete terminal emulation services for a standard ANSI or DEC-VT220 terminal.

This control supports secure connections using the TLS protocol. To establish a secure connection to the server using the Secure Shell (SSH) protocol, use the SocketTools SshClient control.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical

updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

Telnet Protocol Control Properties

Property	Description
AutoResolve	Determines if host names and IP addresses are automatically resolved
Binary	Enable or disable binary input and output
Blocking	Gets and sets the blocking state of the control
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 client certificate
CertificatePassword	Gets and sets the password associated with the client certificate
CertificateStatus	Return the status of the server certificate
CertificateStore	Gets and sets the name of the client 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
HashStrength	Return the length of the message digest that was selected
HostAddress	Gets and sets the IP address of the server
HostName	Gets and sets the name of the server
IsBlocked	Return if the control is blocked performing an operation
IsConnected	Determine if the control is connected to a server
IsInitialized	Determine if the control has been initialized
IsReadable	Return if data can be read from the server without blocking
IsWritable	Return if data can be sent to the server without blocking
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error to occur
LocalEcho	Enable or disable the echoing of characters by the server
Options	Gets and sets the options that are used in establishing a connection
Password	Gets and sets the password for the current user
RemotePort	Gets and sets the port number for a remote connection
Secure	Set or return if a connection to the server is secure
SecureCipher	Return the encryption algorithm used to establish the secure connection with the server
SecureHash	Return the message digest selected when establishing the secure connection with the server
SecureKeyExchange	Return the key exchange algorithm used to establish the secure connection with the server
SecureProtocol	Gets and sets the security protocol used to establish the secure connection with the server

Terminal	Gets and sets the terminal type used by the control
ThrowError	Enable or disable error handling by the container of the control
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
UserName	Gets and sets the current user name
Version	Return the current version of the object

AutoResolve Property

Determines if host names and IP 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 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 **AutoResolve** to False.

Data Type

Boolean

See Also

HostAddress Property, HostName Property

Binary Property

Enable or disable binary input and output.

Syntax

object.**Binary** [= { True | False }]]

Remarks

The **Binary** property enables or disables the exchange of binary data between the client and server. If set to False, all characters have the high-bit stripped off and single linefeed characters are automatically converted to carriage-return/linefeed sequences. The default value is True, which specifies that data is to be received unchanged from the server.

Data Type

Boolean

See Also

LocalEcho Property

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 read data when none has been written, an error is generated. Events such as **OnConnect**, **OnDisconnect**, **OnRead** and **OnWrite** are only fired if the connection is non-blocking.

Data Type

Boolean

See Also

IsBlocked Property, IsReadable Property, IsWritable Property

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 = TelnetClient1.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 client certificate.

Syntax

object.**CertificateName** [= *name*]

Remarks

This property sets the common name or friendly name of the certificate that should be used to establish the connection with the server. 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. If a certificate name is specified, the certificate must have a private key associated with it, otherwise the connection attempt 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 client 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
stCertificateNone	No certificate information is available. A secure connection was not established with the server.
stCertificateValid	The certificate is valid.
stCertificateNoMatch	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.
stCertificateExpired	The certificate has expired and is no longer valid. The client can examine the **CertificateExpires** property to determine when the certificate expired.
stCertificateRevoked	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.
stCertificateUntrusted	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.
stCertificateInvalid	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:

```
TelnetClient1.HostName = strHostName
TelnetClient1.Secure = True
```

```
nError = TelnetClient1.Connect()
If nError > 0 Then
     MsgBox "Unable to connect to server " & strHostName, vbExclamation
     Exit Sub
End If

If TelnetClient1.CertificateStatus <> stCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          TelnetClient1.Disconnect
          Exit Sub
     End If
End If

TelnetClient1.Disconnect
```

See Also

CertificateExpires Property, CertificateIssued Property, CertificateIssuer Property, CertificateSubject Property, Secure Property

CertificateStore Property

Gets and sets the name of the client certificate store or file.

Syntax

object.**CertificateStore** [= *store*]

Remarks

This property sets the name of the certificate store that contains the client certificate that should be used when establishing a secure connection with 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 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

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 = TelnetClient1.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 client certificate.

Syntax

object.**CertificateUser** [= *username*]

Remarks

This property sets the name of the user that owns the client certificate that will be used to establish a secure connection with the server. 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
Text sent and received using a string should be converted using the ANSI code page for the current locale. This is the default encoding type.	
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. It is not recommended that you use this code page unless you know that the remote host is sending text which includes OEM characters.	
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.	
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.	
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.	
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 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

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 server.

Syntax

object.**HostAddress** [= *ipaddress*]

Remarks

The **HostAddress** property can be used to set the IP address for a server 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, HostName Property

HostName Property

Gets and sets the name of the server.

Syntax

object.**HostName** [= *hostname*]

Remarks

The **HostName** property should be set to the name of the server 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

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.

Note that this property will return True if there is ***any*** blocking operation being performed by the application, regardless if the specified control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

Blocking Property, LastError Property

IsConnected Property

Determine if the control is connected to a server.

Syntax

object.**IsConnected**

Remarks

The **IsConnected** read-only property is set to a value of true if the control is connected with a server, otherwise the property has a value of false.

Data Type

Boolean

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

IsReadable Property

Return if data can be read from the server without blocking.

Syntax

object.**IsReadable**

Remarks

The **IsReadable** property returns True if data can be read from the server without blocking. For non-blocking connections, this property can be checked before the application attempts to read the data, preventing an error.

Data Type

Boolean

See Also

IsConnected Property, Read Method, OnRead Event

IsWritable Property

Return if data can be sent to the server without blocking.

Syntax

object.**IsWritable**

Remarks

The **IsWritable** property returns True if data can be written without blocking. For non-blocking connections, this property can be checked before the application attempts to send data to the server, 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 send data without an error. The next operation may result in an **stErrorOperationWouldBlock** or **stErrorOperationInProgress** error. The application must treat these errors as recoverable, and should be prepared to retry operations that result in them.

Data Type

Boolean

See Also

IsReadable Property, SendKey Method, Write Method, OnWrite Event

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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

LocalEcho Property

Enable or disable the echoing of characters by the server.

Syntax

object.**LocalEcho** [= { True | False }]]

Remarks

The **LocalEcho** property enables or disables the echoing of characters by the server. If set to True, the server will be instructed to not echo characters, making the client responsible for displaying user input. The default value for this property is False, specifying that the server should echo user input back to the client.

Data Type

Boolean

See Also

Binary Property

Options Property

Gets and sets the options that are used in establishing a connection.

Syntax

object.**Options** [= *value*]

Remarks

The **Options** property is an integer value which specifies one or more options. The value specified for this property will be used as the default options when connecting to the server. The property value is created by using a bitwise operator with one or more of the following values:

Value	Description
telnetOptionNone	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used.
&H400	telnetOptionTunnel This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
&H800	telnetOptionTrustedSite 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 the TLS protocol.
&H1000	telnetOptionSecure This option specifies the client should attempt to establish a secure connection with the server. Note that the server must support secure connections using the TLS protocol.
&H2000	telnetOptionSecureExplicit This option specifies the client should attempt to establish a secure connection with the server using the START_TLS option. The client initiates a standard connection with the server, then requests a secure connection during the option negotiation process.
&H8000	telnetOptionSecureFallback 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.
&H40000	telnetOptionPreferIPv6	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.

Data Type

Integer (Int32)

See Also

Secure Property, Connect Method

Password Property

Gets and sets the password for the current user.

Syntax

object.**Password** [= *password*]

Remarks

The **Password** property specifies the password used to authenticate the user. This property is used as the default value for the **Login** method if no password is specified as an argument.

Data Type

String

See Also

UserName Property, Connect Method, Login Method

RemotePort Property

Gets and sets the port number for a remote connection.

Syntax

object.**RemotePort** [= *portnumber*]

Remarks

The **RemotePort** property is used to set the port number that the control will use to establish a connection with the server.

Data Type

Integer (Int32)

See Also

HostAddress Property, HostName 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 must 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 strongly recommended that any application that sets this property True use error handling to trap an errors that may occur. If the control 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.

Data Type

Boolean

Example

The following example establishes a secure connection to a server:

```
TelnetClient1.HostName = strHostName
TelnetClient1.RemotePort = 992
TelnetClient1.Secure = True

nError = TelnetClient1.Connect()
If nError > 0 Then
    MsgBox "Unable to connect to server " & strHostName, vbExclamation
    Exit Sub
End If

If TelnetClient1.CertificateStatus <> stCertificateValid Then
     nResult = MsgBox("The server certificate could not be validated" & vbCrLf & _
                      "Are you sure you wish to continue?", vbYesNo)

     If nResult = vbNo Then
          TelnetClient1.Disconnect
          Exit Sub
     End If
End If
```

See Also

CertificateStatus 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
stCipherNone	No cipher has been selected. This is not a secure connection with the server.
stCipherRC2	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.
stCipherRC4	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.
stCipherRC5	The RC5 block cipher was selected. This is a variable key length cipher which supports keys up to 2040 bits, in 8-bit increments.
stCipherDES	The DES (Data Encryption Standard) block cipher was selected. This is a fixed key length cipher using 56-bit keys.
stCipherDES3	The Triple DES block cipher was selected. This cipher encrypts the data three times using different keys, effectively using a 168-bit key length.
stCipherDESX	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.
stCipherAES	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.
stCipherSkipjack	The Skipjack block cipher was selected. This is a fixed key length cipher, using 80-bit keys.
stCipherBlowfish	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
stHashMD5	The MD5 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA1	The SHA-1 algorithm was selected. This algorithm has been deprecated and is no longer considered to be cryptographically secure.
stHashSHA256	The SHA-256 algorithm has been selected.
stHashSHA384	The SHA-384 algorithm has been selected.
stHashSHA512	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
stKeyExchangeNone	No key exchange algorithm has been selected. This is not a secure connection with the server.
stKeyExchangeRSA	The RSA public key exchange algorithm has been selected.
stKeyExchangeKEA	The KEA public key exchange algorithm has been selected. This is an improved version of the Diffie-Hellman public key algorithm.
stKeyExchangeDH	The Diffie-Hellman public key exchange algorithm has been selected.
stKeyExchangeECDH	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

Terminal Property

Gets and sets the terminal type used by the control.

Syntax

object.**Terminal** [= *termtype*]

Remarks

The **Terminal** property specifies the terminal type of the server for display purposes. On UNIX based systems, the terminal name corresponds to a termcap or terminfo entry as set in the TERM environment variable. On Windows based systems which implement the telnet service, this property may be ignored and the server will assume that the client is capable of displaying ANSI escape sequences. On VMS systems, the terminal name should correspond to the terminal type used with the SET TERMINAL/DEVICE command.

If this property is set to an empty string and no terminal type is specified when the **Connect** method is called, a default terminal type named "unknown" will be used. On most UNIX and VMS systems this defines a terminal which is not capable of cursor positioning using control or escape sequences. This terminal type may not be recognized and an error may be displayed when the user logs in indicating that the terminal type is invalid.

Refer to the documentation for the server system to determine what terminal type names are available to you. Remember that on UNIX systems, the terminal type is case-sensitive. Some of the more common terminal types are:

Terminal Type	Description
ansi	This terminal type is usually available on UNIX based servers. This specifies that the client is capable of displaying standard ANSI escape sequences for cursor control.
dumb	This terminal type typically specifies a terminal display which does not support control or escape sequences for cursor positioning. If you do not want escape sequences embedded in the data stream and the server returns an error if the terminal type is not specified, try using this terminal type.
pcansi	This terminal type is usually available on UNIX based servers. This specifies that the client is a using a PC terminal emulator that supports basic ANSI escape sequences for cursor control. This may also enable escape sequences which can set the display colors.
vt100	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT100. This specifies that the client is capable of emulating a DEC VT100 terminal. The VT100 supports many of the same cursor control sequences as an ANSI terminal.
vt220	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT220. This specifies that the client is capable of emulating a DEC VT220 terminal, which is a later version of the VT100.
vt320	This terminal type is usually available on UNIX and VMS based servers. On some VMS systems this string may need to be specified as DEC-VT320.

	This specifies that the client is capable of emulating a DEC VT320 terminal, which is similar to the VT100 and VT220 and provides advanced features such as the ability to set display colors.
xterm	This terminal type is may be available on UNIX based servers which have X Windows installed. This specifies that the client is a using the X Windows xterm emulator which supports standard ANSI escape sequences for cursor control.

Data Type

String

See Also

Login Method

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
TelnetClient1.ThrowError = False
nError = TelnetClient1.Connect(strHostName)

If nError > 0 Then
    MsgBox TelnetClient1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

TelnetClient1.ThrowError = True
TelnetClient1.Connect strHostName

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error.

Note that the **Timeout** property also determines the amount of time the control will spend attempting to connect to a server. If a connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
telnetTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
telnetTraceError	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.
telnetTraceWarning	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.
telnetTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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

UserName Property

Gets and sets the current user name.

Syntax

object.**UserName** [= *username*]

Remarks

The **UserName** property specifies the user that is logging in to the server, and is required for authentication purposes. This property is used as the default value for the **Login** method if no password is specified as an argument.

Data Type

String

See Also

Password Property, Connect Method, Login Method

Version Property

Return the current version of the object.

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

Telnet Protocol Control Methods

Method	Description
Abort	Aborts the current session and terminates the connection
Break	Sends a break signal to the server
Cancel	Cancels the current blocking network operation
Connect	Establish a connection with a server
Disconnect	Terminate the connection with a server
Initialize	Initialize the control and validate the runtime license key
Login	Authenticate the user and log them in to the current session
Read	Return data read from the server
ReadLine	Read a line of text from the server and return it in a string buffer
Reset	Reset the internal state of the control
Search	Search for a specific character sequence in the data stream
SendKey	Send a key code to the server
Uninitialize	Uninitialize the control and release any system resources that were allocated
Write	Write data to the server
WriteLine	Write a line of text to the server

Abort Method

Aborts the current session and terminates the connection.

Syntax

object.**Abort**

Parameters

None.

Return Value

A value of zero is returned if the break signal was sent successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Abort** method sends an abort sequence to the server and the connection to the server is terminated. Once this method returns, the client is no longer connected to the server. If a program is currently executing on the server at the time this function is called, that program may be terminated as a result of the session being aborted. Applications should normally call the **Disconnect** method to gracefully disconnect from the server and should only use this function when the connection must be aborted immediately.

See Also

Break Method, Cancel Method

Break Method

Sends a break signal to the server.

Syntax

object.**Break**

Parameters

None.

Return Value

A value of zero is returned if the break signal was sent successfully. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Break** method sends a signal to the server which may terminate an application that is currently running. The actual response to the break signal depends on the application.

See Also

Abort Method, Cancel 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

Abort Method, Break Method

Connect Method

Establish a connection with a server.

Syntax

object.**Connect**([*RemoteHost*], [*RemotePort*], [*Timeout*], [*Options*])

Parameters

RemoteHost

A string which specifies the host name or IP address of the server. If this argument is not specified, it defaults to the value of the **HostAddress** property if it is defined. Otherwise, it defaults to the value of the **HostName** property.

RemotePort

A number which specifies the port to connect to on the server. If this argument is not specified, it defaults to the value of the **RemotePort** property. A value of zero indicates that the default port number for this service should be used to establish the connection.

Timeout

The number of seconds that the client will wait for a response before failing the operation. If this argument is not specified, the value of the **Timeout** property will be used as the default.

Options

A numeric value which specifies one or more options. If this argument is omitted or a value of zero is specified, a default connection will be established. This argument is constructed by using a bitwise operator with any of the following values:

Value	Description	
telnetOptionNone	No additional options are specified when establishing a connection with the server. A standard, non-secure connection will be used.	
&H400	telnetOptionTunnel	This option specifies that a tunneled TCP connection and/or port-forwarding is being used to establish the connection to the server. This changes the behavior of the client with regards to internal checks of the destination IP address and remote port number, default capability selection and how the connection is established.
&H800	telnetOptionTrustedSite	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 the TLS protocol.
&H1000	telnetOptionSecure	This option specifies the client should attempt to establish a secure connection with the server. Note that the server must

		support secure connections using the TLS protocol.
&H2000	telnetOptionSecureExplicit	This option specifies the client should attempt to establish a secure connection with the server using the START_TLS option. The client initiates a standard connection with the server, then requests a secure connection during the option negotiation process.
&H8000	telnetOptionSecureFallback	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.
&H40000	telnetOptionPreferIPv6	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.

See Also

HostAddress Property, HostName Property, Options Property, RemotePort Property, Disconnect Method, OnConnect Event

Disconnect Method

Terminate the connection with a server.

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

IsConnected Property, Connect Method, OnDisconnect Event

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 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 telnetClient = CreateObject("SocketTools.TelnetClient.11")

nError = telnetClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize Method

Login Method

Authenticate the user and log them in to the current session.

Syntax

object.**Login**([*UserName*], [*Password*])

Parameters

UserName

An optional string argument which specifies the username which should be used to authenticate the client session. If this argument is omitted, the value of the **UserName** property will be used.

Password

An optional string argument which specifies the password which should be used to authenticate the client session. If this argument is omitted, the value of the **Password** 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 **Login** method is specifically designed to work with most UNIX based servers, and may work with other servers that use a similar login process. The method works by scanning the data stream for a username prompt and then replying with the specified username. If that is successful, it will then scan for a password prompt and provide the specified password. If no recognized prompt is found, or if the server responds with an error indicating that the username or password is invalid, the method will fail.

If the **Login** method succeeds, the next call to the **Read** method by the client will return any welcome message to the user. This is typically followed by a command prompt where the user can enter commands to be executed on the server. The data sent by the server during the login process is discarded and not available when the function returns. If the client requires this information, use the **Search** method to automate the login process instead.

Because the **Login** method is designed for UNIX based systems, it may not work with servers running on other operating system platforms such as Windows or VMS. In this case, applications should use the **Search** method to search for the appropriate login prompts in the data stream.

See Also

Connect Method, Disconnect Method, Read Method, Search Method

Read Method

Return data read from the server.

Syntax

object.**Read**(*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, 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 server 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 server, 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 stop 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.

See Also

CodePage Property, IsConnected Property, IsReadable Property, Search Method, Write Method, OnRead Event, OnWrite Event

ReadLine Method

Read up to a line of data from the server and returns it in a string buffer.

Syntax

object.**ReadLine**(*Buffer*, [*Length*])

Parameters

Buffer

A string that the data will be stored in when the method returns. This parameter must be passed by reference.

Length

An optional parameter that specifies the maximum number of bytes to read. If this argument is omitted, then the control will return up to 4096 characters in the string. If the application expects that a single line of text will exceed this value, then it must be explicitly specified.

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 server 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 server 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 network operation is attempted while **ReadLine** is blocked waiting for data from the server, an error will occur. It is recommended that this method only be used with blocking 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, Timeout Property, Read Method, Write Method, WriteLine 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, open network connections will be closed and any handles allocated by the control will be released.

See Also

Cancel Method, Initialize Method, Uninitialize Method

Search Method

Search for a specific character sequence in the data stream.

Syntax

object.**Search**(*String*, [*Buffer*], [*Length*], [*Options*])

Parameters

String

A string argument which specifies the sequence of characters to search for in the data stream. When the control encounters this sequence, the method will return.

Buffer

An optional string or byte array buffer that will contain the output sent by the server, up to and including the search string character sequence. If this argument is omitted, the control will still search for the character sequence but any output sent by the server will be discarded.

Length

An optional integer value which specifies the maximum number of bytes of data to store in the buffer. If this argument is omitted, no limit will be placed on the amount of output buffered by the control.

Options

An optional integer argument which is reserved for future use. This argument should be omitted.

Return Value

This method returns a Boolean value. A return value of true indicates that the search string was found in the data stream. A return value of False indicates that the search string was not found in the amount of time specified by the **Timeout** property or that the server closed the connection.

Remarks

The **Search** method searches for a character sequence in the data stream and stops reading when it is found. This is useful when the client wants to automate responses to the server, such as executing a command and processing the output. The function collects the output from the server and stores it in a buffer provided by the caller. When the function returns, the buffer will contain everything sent by the server up to and including the search string.

See Also

IsReadable Property, Timeout Property, Connect Method, Login Method, Read Method

SendKey Method

Send a key code to the server.

Syntax

object.**SendKey**(*Key*)

Parameters

Key

A value which specifies the key code to send to the server. This may be a single byte, in which case it is sent to the server as-is. If a numeric value is specified, then this is considered to be an ASCII character value and it is sent to the server as a single byte. The value must be between 1 and 255. If the key code value is 0, then the method returns without sending any data. If the value is greater than 255, an error will be raised. If the *Key* argument is a string, then the method will send that string to the server. An empty string is ignored and the method will return without sending any data. An error will be returned if the string is longer than 128 bytes.

Return Value

This method will return a value of true if the key code was successfully sent to the server. If the key cannot be sent, the method will return False and the **LastError** property will contain the error code that indicates the reason for the failure. This method will also return False if the key code value is zero or an empty string is passed by the caller.

Remarks

The **SendKey** method sends a key code to the server. This method is useful if the application needs to send a single character to the server, as opposed to using the **Write** method which should be used for sending large amounts of data.

The strings sent by the **SendKey** method are typically short escape sequences which are generated by a terminal emulator when the user presses a special key, such as a function key. For example, a DEC VT100 terminal sends the escape sequence <ESC>[M when the user presses the F1 function key. To simulate this, those three bytes could be passed as the *Key* value.

Example

The following example demonstrates how to use the **SendKey** method in conjunction with the **KeyMapped** and **KeyPress** events in the Terminal Emulator control:

```
Private Sub Terminal1_KeyMapped(KeyIndex As Integer, Shift As Integer, KeyString As
String)
    TelnetClient1.SendKey KeyString
End Sub

Private Sub Terminal1_KeyPress(KeyAscii As Integer)
    TelnetClient1.SendKey KeyAscii
End Sub
```

See Also

IsWritable Property, Write Method, OnWrite 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 server.

Syntax

object.**Write**(*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 bytes actually written to the server, or -1 if an error was encountered.

Remarks

The **Write** method sends the data in ***buffer*** to the server. 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 blocking, the application will wait until the data can be sent. If the control is non-blocking and the write fails because it could not send all of the data to the server, the **OnWrite** event will be fired when the server can accept data again.

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 ANSI 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, IsConnected Property, IsWritable Property, Timeout Property, Read Method, SendKey Method, OnWrite Event

WriteLine Method

Send a line of text to the server, terminated by a carriage-return and linefeed.

Syntax

object.**WriteLine**([*Buffer*])

Parameters

Buffer

A string which contains the data that will be sent to the server. 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. 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 has been written to the server. If an error occurs, the method will return False.

Remarks

The **WriteLine** method writes a line of text to the server 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.

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 server 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 server, 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

CodePage Property, IsWritable Property, Timeout Property, Read Method, ReadLine Method, Write Method

Telnet Protocol Control Events

Event	Description
OnCancel	This event is generated when a blocking operation is canceled
OnConnect	This event is generated when a connection is established
OnDisconnect	This event is generated when a connection is terminated
OnError	This event is generated when a control error occurs
OnRead	This event is generated when data is available to be read
OnTimeout	This event is generated when a blocking operation times out
OnWrite	This event is generated when data can be written to the server

OnCancel Event

The **OnCancel** event is generated when a blocking operation is canceled.

Syntax

Sub *object*_**OnCancel** ([*Index* **As Integer**])

Remarks

This event is generated 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, OnTimeout 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 server as a result of a **Connect** method call. This event is only triggered when the **Blocking** property is set to False.

See Also

Blocking Property, Connect Method, OnDisconnect Event, OnWrite 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. This event is only triggered when the **Blocking** property is set to False.

When the **OnDisconnect** event fires, it is possible that there may still be buffered data available to read from the server. Before disconnecting from the server, the application should attempt to read any remaining data until the **Read** method returns a value of zero, or returns an error indicating that the operation would block.

See Also

Blocking Property, IsConnected Property, IsReadable Property, Connect Method, Disconnect Method, Read Method, 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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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

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 triggered when the **Blocking** property is set to False.

See Also

IsReadable Property, 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. This event is only triggered when the **Blocking** property is set to True.

See Also

Timeout Property, OnCancel Event

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 triggered when the **Blocking** property is set to False.

See Also

IsWritable Property, Read Method, SendKey Method, Write Method, OnConnect Event, OnRead Event

Terminal Emulation Control

Emulate an ANSI or DEC VT-220 character mode display terminal.

Reference

- Properties
- Methods
- Events

Control Information

Object Name	TerminalCtl.Terminal
File Name	CSNVTX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.Terminal.11
ClassID	BD5466AA-B72B-4BB0-824B-F5F8F70DBE01
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None

Remarks

The Terminal Emulation control provides a comprehensive interface for emulating an ANSI or DEC-VT220 terminal, with full support for all standard escape and control sequences, color mapping and other advanced features. The control provides both a high level interface for parsing escape sequences and updating a display, as well as lower level primitives for directly managing the virtual display, such as controlling the individual display cells, moving the cursor position and specifying display attributes.

This control can be used in conjunction with the Remote Command, Secure Shell or Telnet Protocol controls to provide terminal emulation services for an application, or it can be used independently. For example, this control could be used to provide emulation services for a program that connects to a device using an RS-232 serial port.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

Distribution

When you distribute an application that uses this control, you should install the appropriate file in the Windows system directory. ActiveX controls must be registered on the target system by the installation program before they can be used by an application.

Terminal Emulator Control Properties

Property	Description
Attributes	Gets and sets the current display attribute for the terminal emulator
AutoRefresh	Enable or disable the automatic refreshing of the virtual display
AutoSelect	Enable or disable the automatic selection of text in the display
AutoWrap	Enable or disable the wrapping of text in the emulation window
BackColor	Sets or returns the background color for the control.
Bell	Enable or disable the audible bell
BoldColor	Sets or returns the bold color for the control.
Cell	Returns information about the specified character cell in the display
CellHeight	Return the height of a text cell
CellWidth	Return the width of a text cell
CodePage	Gets and sets the code page used when reading and writing text
ColorMap	Gets and sets the RGB value used when displaying color text attributes
Columns	Gets and sets the number of columns in the emulation display
Cursor	Enable or disable the display of the cursor in the emulation window
CursorStyle	Gets and sets the style of cursor used in the emulator
CursorX	Gets and sets the current cursor position in the display
CursorY	Gets and sets the current cursor position in the display
Emulation	Gets and sets the emulation used by the control
Font	Returns the Font object used by the terminal emulator
FontBold	Gets and sets the bold style for the current font
FontName	Gets and sets the name of the current font
FontSize	Gets and sets the point size of the current font
ForeColor	Sets or returns the foreground color for the control.
hWnd	Returns a handle to the control window
KeyMap	Gets and sets the character sequence mapped to a special key
MouseX	Return the current mouse pointer position in the display
MouseY	Return the current mouse pointer position in the display
MousePointer	Gets and sets the type of pointer which is displayed when the mouse is positioned over the control window
NewLine	Determine how carriage returns and linefeeds are displayed
Rows	Gets and sets the number of rows in the emulation display
ScrollBars	Returns or sets a value indicating whether the control has horizontal or vertical scroll bars
SelLength	Gets and sets the number of characters selected
SelStart	Gets and sets the starting position of the current text selection
SelText	Returns the selected text or text from a specific portion of the display
Text	Gets and sets the text displayed by the control
Version	Return the current version of the object

Attributes Property

Gets and sets the current display attribute for the terminal emulator.

Syntax

object.**Attributes** [= *attributes*]

Remarks

The **Attributes** property can be used to determine the current display attributes, or to change the current attribute for subsequent text. The following table lists the attributes that are recognized by the control.

Value	Description
nvtAttributeNormal	Normal, default attributes.
nvtAttributeReverse	Foreground and background cell colors are reversed.
nvtAttributeBold	The character is displayed using a higher intensity color.
nvtAttributeDim	The character is displayed using a lower intensity color.
nvtAttributeUnderline	The character is displayed with an underline.
nvtAttributeHidden	The character is stored in display memory, but not shown.
nvtAttributeProtect	The character is protected and cannot be cleared.

Data Type

Integer (Int32)

See Also

BoldColor Property, Cell Property, CursorStyle Property

AutoRefresh Property

Enable or disable the automatic refreshing of the virtual display.

Syntax

object.**AutoRefresh** [= { True | False }]]

Remarks

The **AutoRefresh** property is used to enable or disable the automatic refreshing of the virtual display whenever characters are written or the cursor position changes. By setting the property to False, the display can be changed and those changes will not be displayed until the property is reset to True. This allows an application to make a series of changes to the display text, attributes or cursor position without causing it to flicker.

Data Type

Boolean

See Also

Attributes Property, Refresh Method, Write Method

AutoSelect Property

Enable or disable the automatic selection of text in the display.

Syntax

object.**AutoSelect** [= { True | False }]]

Remarks

The **AutoSelect** property is used to enable or disable the automatic selection of text in the virtual display. When the property is set to True, the user can select text by clicking and dragging the mouse over the text to be selected. When set to False, no text is selected if the user drags the mouse over the display.

Data Type

Boolean

See Also

SelText Property, Select Method

AutoWrap Property

Enable or disable the wrapping of text in the emulation window.

Syntax

object.**AutoWrap** [= { True | False }]]

Remarks

The AutoWrap property enables or disables the wrapping of text in the emulation window. If set to True, when text reaches last column in the display, the cursor is re-positioned to the first column on the next line. If set to False, any text displayed beyond the last column is discarded.

Data Type

Integer (Int32)

See Also

Bell Property, Columns Property, Emulation Property, Rows Property

BackColor Property

Sets or returns the background color for the control.

Syntax

object.**BackColor** [= *color*]

Remarks

The **BackColor** property returns the current background color for the control. Setting the property changes the color to the specified value.

Colors are RGB (Red Green Blue) values which range from 0 to 16,777,215 (&HFFFFFF). The high byte of a number in this range equals 0; the lower 3 bytes, from least to most significant byte, determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number between 0 and 255 (&HFF). If the high byte isn't 0, the control uses the system colors, as defined in the user's Control Panel settings.

The following table lists the standard system color values and the Control panel settings that they correspond to:

Color Value	Definition	Control Panel Setting
80000000h	Scrollbar background	
80000001h	Desktop	Desktop
80000002h	Active window caption	Active Title Bar
80000003h	Inactive window caption	Inactive Title Bar
80000004h	Menu background	Menu
80000005h	Window background	Window
80000006h	Window frame	
80000007h	Menu text	Menu
80000008h	Window text	Window
80000009h	Window caption text	Active Title Bar
8000000Ah	Active window border	Active Window Border
8000000Bh	Inactive window border	Inactive Window Border
8000000Ch	Background color	Application Background
8000000Dh	Items selected in a control	Selected Items
8000000Eh	Text of items selected in a control	Selected Items
8000000Fh	Face shading on push buttons	
80000010h	Edge shading on push buttons	
80000011h	Disabled text	
80000012h	Text on push buttons	Caption Buttons
80000013h	Text color for an inactive caption	Inactive Title Bar
80000014h	Highlight color for buttons	

80000015h	Dark color for 3D display elements	
80000016h	Light color for 3D display elements	
80000017h	Text color for ToolTip controls	ToolTip
80000018h	Background color for ToolTip controls	ToolTip

Data Type

Integer (Int32)

See Also

BoldColor Property, ColorMap Property, ForeColor Property

Bell Property

Enable or disable the audible bell.

Syntax

object.**Bell** [= { True | False }]]

Remarks

The **Bell** property enables or disables the audible bell which is played whenever the control character Ctrl+G is encountered. The default property value is True.

Data Type

Boolean

See Also

AutoWrap Property, Cursor Property

BoldColor Property

Sets or returns the bold color for the control.

Syntax

object.**BoldColor** [= *color*]

Remarks

The **BoldColor** property returns the current color used for bold text in the control. Setting the property changes the color to the specified value.

Colors are RGB (Red Green Blue) values which range from 0 to 16,777,215 (&HFFFFFF). The high byte of a number in this range equals 0; the lower 3 bytes, from least to most significant byte, determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number between 0 and 255 (&HFF). If the high byte isn't 0, the control uses the system colors, as defined in the user's Control Panel settings.

The following table lists the standard system color values and the Control panel settings that they correspond to:

Color Value	Definition	Control Panel Setting
80000000h	Scrollbar background	
80000001h	Desktop	Desktop
80000002h	Active window caption	Active Title Bar
80000003h	Inactive window caption	Inactive Title Bar
80000004h	Menu background	Menu
80000005h	Window background	Window
80000006h	Window frame	
80000007h	Menu text	Menu
80000008h	Window text	Window
80000009h	Window caption text	Active Title Bar
8000000Ah	Active window border	Active Window Border
8000000Bh	Inactive window border	Inactive Window Border
8000000Ch	Background color	Application Background
8000000Dh	Items selected in a control	Selected Items
8000000Eh	Text of items selected in a control	Selected Items
8000000Fh	Face shading on push buttons	
80000010h	Edge shading on push buttons	
80000011h	Disabled text	
80000012h	Text on push buttons	Caption Buttons
80000013h	Text color for an inactive caption	Inactive Title Bar
80000014h	Highlight color for buttons	

80000015h	Dark color for 3D display elements	
80000016h	Light color for 3D display elements	
80000017h	Text color for ToolTip controls	ToolTip
80000018h	Background color for ToolTip controls	ToolTip

Data Type

Integer (Int32)

See Also

BackColor Property, ColorMap Property, ForeColor Property

Cell Property

Returns information about the specified character cell in the display.

Syntax

object.**Cell**(*X*, *Y*)

Remarks

The **Cell** property returns information about the specified character cell in the display at the specified X and Y cursor position.

The value returned by the **Cell** property is a 32-bit integer value, where the low order word specifies the ANSI character stored at that position and the high order word specifies the display attributes for that cell.

The character cell attributes may be one or more of the following values:

Value	Description
nvtAttributeNormal	Normal, default attributes.
nvtAttributeReverse	Foreground and background cell colors are reversed.
nvtAttributeBold	The character is displayed using a higher intensity color.
nvtAttributeDim	The character is displayed using a lower intensity color.
nvtAttributeUnderline	The character is displayed with an underline.
nvtAttributeHidden	The character is stored in display memory, but not shown.
nvtAttributeProtect	The character is protected and cannot be cleared.

One or more attributes may be combined using a bitwise Or operator. Certain attributes, such as **nvtAttributeBold** and **nvtAttributeDim** are mutually exclusive.

Data Type

Integer (Int32)

Example

To access the high and low words of the value returned by the **Cell** property in Visual Basic 6, it's useful to implement two helper functions:

```
Public Function LoWord(DWord As Long) As Integer
    If DWord And &H8000& Then
        LoWord = DWord Or &HFFFF0000
    Else
        LoWord = DWord And &HFFFF&
    End If
End Function

Public Function HiWord(DWord As Long) As Integer
    HiWord = (DWord And &HFFFF0000) \ &H10000
End Function
```

You can then use those functions to get the character and attributes for the cell:

```
Dim nCell As Long
Dim nChar As Integer
Dim nAttribute As Integer
```

```
nCell = Terminal1.Cell(xPos, yPos)
nChar = LoWord(nCell)
nAttribute = HiWord(nCell)
```

See Also

Attributes Property

CellHeight Property

Return the height of a text cell.

Syntax

object.**CellHeight**

Remarks

The **CellHeight** property returns the height of a text cell in pixels. This value can be used in calculating the minimum height of the display window in order to display all lines of text.

Data Type

Integer (Int32)

See Also

CellWidth Property

CellWidth Property

Return the width of a text cell.

Syntax

object.**CellWidth**

Remarks

The **CellWidth** property returns the width of a text cell in pixels. This value can be used in calculating the minimum width of the display window in order to display all text columns.

Data Type

Integer (Int32)

See Also

CellHeight 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
Text sent and received using a string should be converted using the ANSI code page for the current locale. This is the default encoding type.	
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. It is not recommended that you use this code page unless you know that the remote host is sending text which includes OEM characters.	
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.	
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.	
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.	
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.

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 which are written to the virtual display. Similarly, when copying characters from the display into a string buffer, those characters are converted to Unicode before they are returned to your application.

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

SelText Property, Text Property, Write Method

ColorMap Property

Gets and sets the RGB value used when displaying color text attributes.

Syntax

object.**ColorMap**(*Index*) [= *color*]

Remarks

The **ColorMap** property array provides access to the virtual display color table which determines what RGB values are used to display foreground and background text color attributes.

When the emulator processes an escape sequence that changes the current foreground or background color, the actual RGB color value is determined by looking up the value in the virtual display's color table. The **ColorMap** property is useful for determining what values are being used when a color attribute is set and enables an application to change those colors. The emulator currently supports a maximum of sixteen (16) color values, and the index into the table corresponds to the color as defined by the standard for ANSI terminals:

Index	Color	Default (Hex)	Default (Integer)	Default (RGB)
Black	RGB(0,0,0)			
Red	000000A0h	RGB(160,0,0)		
Green	0000A000h	RGB(0,160,0)		
Yellow	0000A0A0h	RGB(160,160,0)		
Blue	00A00000h	RGB(0,0,160)		
Magenta	00A000A0h	RGB(160,0,160)		
Cyan	00A0A000h	RGB(0,160,160)		
White	00E0E0E0h	RGB(224,224,224)		
Gray	00C0C0C0h	RGB(192,192,192)		
Light Red	008080FFh	RGB(255,128,128)		
Light Green	0090EE90h	RGB(144,238,144)		
Light Yellow	00C0FFFFh	RGB(255,255,192)		
Light Blue	00E6D8ADh	RGB(173,216,230)		
Light Magenta	00FFC0FFh	RGB(255,192,255)		
Light Cyan	00FFFFE0h	RGB(224,255,255)		
High White	00FFFFFFh	RGB(255,255,255)		

A standard ANSI color terminal supports eight standard colors (0-7). To select a foreground color, you add 30 to the color index and pass that value as a parameter to the SGR (select graphic rendition) escape sequence. To select a background color, you add 40 to the color index. For example, to set the current foreground color to white and the background color to blue, you could send the following escape sequence:

```
ESC [ 37;44 m
```

Note that if you wanted to set the foreground color to a bold version of standard yellow, you

would first set the bold attribute, and then use the index value of 3, such as:

```
ESC [ 1;33m
```

Changing the value of the **ColorMap** property array allows the application to make selective changes to the actual RGB color value that is used when a color attribute is set. Note that changes to the color map will only affect new characters as they are displayed, not any previously displayed characters.

Data Type

Integer (Int32)

See Also

BackColor Property, BoldColor Property, Cell Property, ForeColor Property

Columns Property

Gets and sets the number of columns in the emulation display.

Syntax

object.**Columns** [= *columns*]

Remarks

The **Columns** property returns the number of columns in the emulation window, or allows the application to change the number of columns. Currently, the number of columns may only be set to 80 or 132. Note that changing the number of columns in the display causes the current display to be invalidated, and the window will be cleared.

Data Type

Integer (Int32)

See Also

CellHeight Property, CellWidth Property, Rows Property, ScrollBars Property

Cursor Property

Enable or disable the display of the cursor in the emulation window.

Syntax

object.**Cursor** [= { True | False }]]

Remarks

The **Cursor** property enables or disables the display of the cursor in the emulation window. The default property value is True.

Data Type

Boolean

See Also

CursorStyle Property, Emulation Property

CursorStyle Property

Gets and sets the style of cursor used in the emulator.

Syntax

object.**CursorStyle** [= *style*]

Remarks

The **CursorStyle** property determines how the cursor is displayed in the emulation window. The following values may be used.

Value	Description
nvtUnderline	The cursor is displayed as an underline
nvtBlock	The cursor is displayed as a block that is the full height of the character cell

Data Type

Integer (Int32)

See Also

Cursor Property

CursorX Property

Gets and sets the current cursor position in the display.

Syntax

object.**CursorX** [= *column*]

Remarks

The **CursorX** property returns the current position of the cursor in the display, or can be used to change the current position. The current position is given in columns and indicates where the next text character will be displayed. To calculate the pixel offset where the cursor is located in the control window, multiply this value by the **CellWidth** property value.

Data Type

Integer (Int32)

See Also

CellWidth Property, Cursor Property, CursorStyle Property, CursorY Property, MouseX Property, MouseY Property

CursorY Property

Gets and sets the current cursor position in the display.

Syntax

object.**CursorY** [= *row*]

Remarks

The **CursorY** property returns the current position of the cursor in the display, or can be used to change the current position. The current position is given in rows and indicates where the next text character will be displayed. To calculate the pixel offset where the cursor is located in the control window, multiply this value by the **CellHeight** property value.

Data Type

Integer (Int32)

See Also

CellHeight Property, Cursor Property, CursorStyle Property, CursorX Property

Emulation Property

Gets and sets the emulation used by the control.

Syntax

object.**Emulation** [= *type*]

Remarks

The **Emulation** property can be used to set or return the type of emulation performed by the control. The following values may be used:

Value	Description
nvtNone	The virtual display does not emulate any specific terminal type, and does not process any escape sequences.
nvtANSI	The virtual display processes ANSI escape sequences for screen management and cursor positioning. This emulation also supports escape sequences to control the foreground and background color. The default keymap for ANSI function key escape sequences will be selected. This is the default value.
nvtVT100	The virtual display processes DEC VT-100 escape sequences for screen management and cursor positioning. The default keymap for a DEC VT-100 terminal will be selected.
nvtVT220	The virtual display processes DEC VT-220 escape sequences for screen management and cursor positioning. This emulation also supports DEC VT-320 escape sequences to control the foreground and background color. The default keymap for a DEC VT-220 terminal will be selected.

Data Type

Integer (Int32)

See Also

AutoWrap Property, Cursor Property, Write Method

Font Property

Returns the Font object used by the terminal emulator.

Syntax

object.**Font**

Remarks

The **Font** property returns the Font object which is used by the control. The Font object determines the font name and attributes which are used when drawing text in the virtual display. The control expects that the font will be fixed-width, where the width and height of each character is the same. The use of a variable width font may cause the control to display the cursor in the wrong location.

By default, the control will use the standard Terminal font, which is a fixed-width OEM font that is suitable for most applications.

Data Type

Font Object

Example

The following example demonstrates how to use the **Font** property to change the name of the font used by the control:

```
Terminal1.Font.Name = "Lucida Console"
```

See Also

FontBold Property, FontName Property, FontSize Property

FontBold Property

Gets and sets the bold style for the current font.

Syntax

object.**FontBold** [= { True | False }]

Remarks

The **FontBold** property returns True if the current font style has the bold attribute enabled. Setting this property changes the style for the current font. Changing the **Bold** property of the object returned by the **Font** property will automatically update this property.

This property is provided for compatibility with languages which do not support the standard Font object, or where using the Font object interface is difficult. When possible, it is recommended that programs use the **Font** property instead.

Data Type

Boolean

See Also

Font Property, FontName Property, FontSize Property

FontName Property

Gets and sets the name of the current font.

Syntax

object.**FontName** [= *fontname*]

Remarks

The **FontName** property returns the name of the current font. Setting this property changes the font which is used by the control to display text. Changing the **Name** property of the object returned by the **Font** property will automatically update this property.

The control expects that the font will be fixed-width, where the width and height of each character is the same. The use of a variable width font may cause the control to display the cursor in the wrong location.

This property is provided for compatibility with languages which do not support the standard Font object, or where using the Font object interface is difficult. When possible, it is recommended that programs use the **Font** property instead.

Data Type

String

See Also

Font Property, FontBold Property, FontSize Property

FontSize Property

Gets and sets the point size of the current font.

Syntax

object.**FontSize** [= *fontsize*]

Remarks

The **FontSize** property returns the size of the current font. Setting this property changes the size of the font which the control uses to display text. Changing the **Size** property of the object returned by the **Font** property will automatically update this property.

This property is provided for compatibility with languages which do not support the standard Font object, or where using the Font object interface is difficult. When possible, it is recommended that programs use the **Font** property instead.

Data Type

Integer (Int32)

See Also

Font Property, FontBold Property, FontName Property

ForeColor Property

Sets or returns the foreground color for the control.

Syntax

object.**ForeColor** [= *color*]

Remarks

The **ForeColor** property returns the current foreground color for the control. Setting the property changes the color to the specified value.

Colors are RGB (Red Green Blue) values which range from 0 to 16,777,215 (&HFFFFFF). The high byte of a number in this range equals 0; the lower 3 bytes, from least to most significant byte, determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number between 0 and 255 (&HFF). If the high byte isn't 0, the control uses the system colors, as defined in the user's Control Panel settings.

The following table lists the standard system color values and the Control panel settings that they correspond to:

Color Value	Definition	Control Panel Setting
80000000h	Scrollbar background	
80000001h	Desktop	Desktop
80000002h	Active window caption	Active Title Bar
80000003h	Inactive window caption	Inactive Title Bar
80000004h	Menu background	Menu
80000005h	Window background	Window
80000006h	Window frame	
80000007h	Menu text	Menu
80000008h	Window text	Window
80000009h	Window caption text	Active Title Bar
8000000Ah	Active window border	Active Window Border
8000000Bh	Inactive window border	Inactive Window Border
8000000Ch	Background color	Application Background
8000000Dh	Items selected in a control	Selected Items
8000000Eh	Text of items selected in a control	Selected Items
8000000Fh	Face shading on push buttons	
80000010h	Edge shading on push buttons	
80000011h	Disabled text	
80000012h	Text on push buttons	Caption Buttons
80000013h	Text color for an inactive caption	Inactive Title Bar
80000014h	Highlight color for buttons	

80000015h	Dark color for 3D display elements	
80000016h	Light color for 3D display elements	
80000017h	Text color for ToolTip controls	ToolTip
80000018h	Background color for ToolTip controls	ToolTip

Data Type

Integer (Int32)

See Also

BackColor Property, BoldColor Property, ColorMap Property

hWnd Property

Returns a handle to the control window.

Syntax

object.**hWnd**

Remarks

The **hWnd** property returns the handle to the control window. The Windows operating system identifies each form or control in an application by assigning it a numeric value called a handle, or hWnd. This handle is required by many Windows API functions which can be used to control the behavior and appearance of a window. For more information, refer to the Windows User Interface documentation or the Windows API technical reference which is part of the Microsoft Windows SDK.

Note that because the **hWnd** property value can change while the program is running, you should never store the value in a variable.

Data Type

Integer (Int32)

KeyMap Property

Gets and sets the character sequence mapped to a special key.

Syntax

object.**KeyMap**(*Index*) [= *value*]

Remarks

The **KeyMap** property array allows the application to define character sequences that should be mapped to special keys. When a special key is pressed in the emulation window and there is an entry for it in the key map, the **KeyMapped** event is fired.

The property array index identifies the key which will be mapped. Refer to the KeyMap Constants table for a list of keys and their corresponding values which may be mapped by the application.

Data Type

String

See Also

KeyMapped Event

MousePointer Property

Gets and sets the type of pointer which is displayed when the mouse is positioned over the control window.

Syntax

object.**MousePointer** [= *pointer*]

Remarks

The **MousePointer** property returns the type of pointer that is currently displayed when the mouse is positioned over the control window. Setting the property changes the type of mouse pointer that is displayed. This can be useful if you wish to indicate a change in functionality, such as changing the pointer to an hourglass when the program is busy processing information.

The following values may be assigned to this property:

Value	Description
nvtDefault	The default mouse pointer
nvtArrow	Arrow pointer
nvtCrosshair	Crosshair pointer
nvtIBeam	I-beam pointer, commonly used when editing text
nvtIcon	Icon pointer, a small square within a square
nvtSize	Size pointer, four point arrow pointing north, south, east and west
nvtSizeNESW	Size pointer, arrows pointing northeast and southwest
nvtSizeSN	Size pointer, arrows pointing north and south
nvtSizeNWSE	Size pointer, arrows pointing northwest and southeast
nvtSizeWE	Size pointer, arrows pointing west and east
nvtUpArrow	Up arrow pointer
nvtHourglass	Hourglass pointer
nvtNoDrop	No drop pointer, a circle with a slash through it
nvtArrowHourglass	Arrow pointer with a small hourglass
nvtArrowQuestion	Arrow pointer with a small question mark
nvtArrowSizeAll	Size all pointer

Data Type

Integer (Int32)

Example

The following example sets the MousePointer to an hourglass, which is typically used to indicate to the user that they program is busy and that they should wait for it to complete the current operation:

```
Terminal1.MousePointer = nvtHourglass
```

See Also

MouseMove Event

MouseX Property

Return the current mouse pointer position in the display.

Syntax

object.**MouseX**

Remarks

The **MouseX** property returns the current position of the mouse pointer in the display. The current position is given in columns, not pixels. To calculate the pixel offset where the mouse pointer is located in the control window, multiply this value by the **CellWidth** property value.

Data Type

Integer (Int32)

See Also

CellWidth Property, CursorX Property, CursorY Property, MouseY Property

MouseY Property

Return the current mouse pointer position in the display.

Syntax

object.**MouseY**

Remarks

The **MouseY** property returns return the current position of the mouse pointer in the display. The current position is given in rows, not pixels. To calculate the pixel offset where the mouse pointer is located in the control window, multiply this value by the **CellHeight** property value.

Data Type

Integer (Int32)

See Also

CellHeight Property, CursorX Property, CursorY Property, MouseX Property

NewLine Property

Determine how carriage returns and linefeeds are displayed.

Syntax

object.**NewLine** [= *newline*]

Remarks

The **NewLine** property controls how carriage returns and linefeeds are processed by the emulator. The following values may be used:

Value	Description
nvtCRLF	A carriage return positions the cursor to the first column, and a linefeed advances the cursor to the next row, scrolling the display if necessary. This is the default value.
nvtCR	A carriage return positions the cursor to the first column and advances to the next row, scrolling the display if necessary.
nvtLF	A linefeed positions the cursor to the first column and advances to the next row, scrolling the display if necessary.

Data Type

Integer (Int32)

See Also

AutoWrap Property, Columns Property, Write Method

Rows Property

Gets and sets the number of rows in the emulation display.

Syntax

object.**Rows** [= *rows*]

Remarks

The **Rows** property returns the number of rows in the emulation window, or allows the application to change the number of rows. Changing the number of rows in the display causes the current display to be invalidated, and the window will be cleared.

Data Type

Integer (Int32)

See Also

CellHeight Property, CellWidth Property, Columns Property, ScrollBars Property

ScrollBars Property

Description

Returns or sets a value indicating whether the control has horizontal or vertical scroll bars.

Syntax

object.**ScrollBars** [= *bartype*]

Remarks

The **ScrollBars** property determines what kind of scroll bars are displayed if the virtual display is larger than the emulation control's window. It may be one of the following values:

Value	Description
nvtSBNone	Do not display scrollbars
nvtHorizontal	Display a horizontal scrollbar if necessary
nvtVertical	Display a vertical scrollbar if necessary
nvtBoth	Display both horizontal and vertical scrollbars if necessary

Scroll bars are only displayed if needed. If the emulation window is large enough to display all of the columns and rows, no scrollbars will be drawn even if they are enabled using this property.

Data Type

Integer (Int32)

See Also

CellHeight Property, CellWidth Property, Columns Property, Rows Property

SelLength Property

Gets and sets the number of characters selected.

Syntax

object.**SelLength** [= *length*]

Remarks

The **SelLength** property is used to return the number of characters currently selected in the emulation window. When used in conjunction with the **SelStart** property, it can be used to select text from the display.

Data Type

Integer (Int32)

See Also

AutoSelect Property, SelStart Property, SelText Property, Text Property, Deselect Method, Select Method

SelStart Property

Gets and sets the starting position of the current text selection.

Syntax

object.**SelStart** [= *offset*]

Remarks

The **SelStart** property specifies an offset which is the starting position of the selected text. This property can be used in conjunction with the **SelLength** property to select text in the virtual display.

To convert the cursor position to an offset, multiply the y-position by the number of columns and add the x-position. The **SelLength** property determines the number of characters to copy from the starting position. Reading the **SelText** property returns the text displayed at the selected location.

Data Type

Integer (Int32)

See Also

SelLength Property, SelText Property, Text Property, Deselect Method, Select Method

SelText Property

Returns the selected text or text from a specific portion of the display.

Syntax

object.**SelText**

Remarks

The **SelText** property returns the text which is currently selected in the virtual display. If no text has been selected and the **SelLength** property is greater than zero, then the text starting at the position specified by the **SelStart** property will be returned.

To read a single character a specific location in the display, it is preferable to use the **Cell** property rather than calculating the offset, setting the **SelStart** property and then reading the **SelText** property.

Data Type

String

See Also

AutoSelect Property, Cell Property, CodePage Property, SelLength Property, SelStart Property, Text Property, Deselect Method, Select Method

Text Property

Gets and sets the text displayed by the control.

Syntax

object.**Text** [= *value*]

Remarks

The **Text** property returns the text displayed by the control. The string that is returned contains each row of text in the display, terminated with a carriage-return linefeed. Empty cells at the end of a row are ignored so there are no extraneous spaces in the text. In other words, the value returned by the **Text** property is similar to how text is returned from a multi-line edit control. If you need to access a character at a specific location in the display, use the **Cell** property. If you need to access the text in a specific part of the display, including any empty cells, then use the **SelStart**, **SelLength** and **SelText** properties instead.

Setting the **Text** property will cause the current contents of the display to be replaced by the contents of the specified string. If the string contains escape sequences and/or control characters, they will be processed according to how the **Emulation** property is set.

Data Type

String

See Also

Cell Property, CodePage Property, Emulation Property, SelLength Property, SelStart Property, SelText Property, Write Method

Version Property

Return the current version of the object.

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

Terminal Emulator Control Methods

Method	Description
Clear	Clear the terminal emulation window
ClearEol	Erase all characters from the current column to the end of the line
DelLine	Delete the current line in the terminal emulation display
Deselect	Deselects any selected text in the display
Initialize	Initialize the control and validate the runtime license key
InsLine	Insert an empty line at the current position in the terminal emulation display
Refresh	Forces a complete redraw of the virtual display
Reset	Reset the internal state of the control
ScrollDown	Scroll the display down by one line
ScrollUp	Scroll the display up by one line
Select	Selects a region of the virtual display and returns the selected text
Uninitialize	Uninitialize the control
Write	Write data to virtual display

CellAttributes Method

Return the display attributes for the terminal emulator for a specified cell.

Syntax

object.**Attribute**(*X*, *Y*)

Parameters

X

An integer value that specifies a horizontal position in the virtual display buffer. This value must be in the range of 0 through the number of **Columns** -1.

Y

An integer value that specifies a vertical position in the virtual display buffer. This value must be in the range of 0 through the number of **Rows** -1.

Return Value

If an invalid position is specified, the method returns -1. Otherwise, the method returns a mask of graphic attributes for the display position. The following table lists the attributes that are recognized by the control. The value returned by the method is a combination of the bits that correspond to these values.

Value	Description
Normal display, no attributes enabled	
Reverse video attribute	
Bold attribute (use BoldColor color to display text)	
Dim attribute	
Blink attribute (not supported)	
Underline attribute	
Hidden attribute	
Protected attribute (not supported)	
Graphics attribute	

See Also

Attributes Property, Columns Property, Rows Property

Clear Method

Clear the terminal emulation window.

Syntax

object.**Clear**

Parameters

None.

Return Value

None.

See Also

ClearEol Method, DelLine Method, InsLine Method, Write Method

ClearEol Method

Erase all characters from the current column to the end of the line.

Syntax

object.**ClearEol**

Parameters

None.

Return Value

None.

See Also

Clear Method, DelLine Method, InsLine Method, Write Method

DelLine Method

Delete the current line in the terminal emulation display.

Syntax

object.**DelLine**

Parameters

None.

Return Value

None.

See Also

Clear Method, ClearEol Method, InsLine Method, Write Method

Deselect Method

Deselects any selected text in the display.

Syntax

object.**Deselect**

Parameters

None.

Return Value

None.

Remarks

The **Deselect** method deselects any text that has been previously selected.

See Also

AutoSelect Property, SelText Property, Select 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 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.

See Also

Uninitialize Method

InsLine Method

Insert an empty line at the current position in the terminal emulation display.

Syntax

object.**InsLine**

Parameters

None.

Return Value

No value is returned by this method.

See Also

Clear Method, ClearEol Method, DelLine Method, Write Method

Refresh Method

Forces a complete redraw of the virtual display.

Syntax

object.**Refresh**

Parameters

None.

Return Value

None.

Remarks

The **Refresh** method forces the control to redraw the virtual display. Normally, the virtual display is automatically redrawn after the display has been modified and there are no other events being processed. However, there may be situations where you want the display updated immediately.

To prevent the control from automatically redrawing when the virtual display has been modified, set the **AutoRefresh** property to False. You can then call the **Refresh** method to force the control to be redrawn as needed.

See Also

AutoRefresh Property

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 and the virtual display is recreated.

See Also

Initialize Method, Uninitialize Method

ScrollDown Method

Scroll the display down by one line.

Syntax

object.**ScrollDown**

Parameters

None.

Return Value

None.

See Also

DelLine Method, InsLine Method, ScrollUp Method

ScrollUp Method

Scroll the display up by one line.

Syntax

object.**ScrollUp**

Parameters

None.

Return Value

None.

See Also

DelLine Method, InsLine Method, ScrollDown Method

Select Method

Selects a region of the virtual display and returns the selected text.

Syntax

object.**Select**([*Column1*], [*Row1*], [*Column2*], [*Row2*], [*Options*)

Parameters

Column1

A optional integer which specifies the starting column for the text selection. If this argument is omitted, the first column in the display is used.

Row1

A optional integer which specifies the starting row for the text selection. If this argument is omitted, the first row in the display is used.

Column2

A optional integer which specifies the ending column for the text selection. If this argument is omitted, the last column in the display is used.

Row2

A optional integer which specifies the ending row for the text selection. If this argument is omitted, the last row in the display is used.

Options

An optional integer value which specifies one or more options. More than one option can be combined using a bitwise operator. The following values may be used:

Value	Description
nvtSelectDefault	The default selection option. If there is a region of the display already selected, it will be cleared and the new region is selected. The selected text is buffered and can be accessed using the **SelText** property.
nvtSelectClipboard	Copy the selected text to the clipboard. If this option is not specified, the selected text is buffered and may be accessed using the **SelText** property.
&H1000	nvtSelectNoRefresh	The display is not refreshed when the region is selected. This is useful if the application is going to be selecting multiple regions of the display, or combining more than one region, in order to minimize output to the window.
&H2000	nvtSelectNoBuffer	Do not buffer the text in the selected region of the display. The display will

		show any text as being selected, but it will not be available to the application. This can be useful if the application is going to select multiple regions and combine them.
&H4000	nvtSelectCombine	If there is already a region of the display that has been selected, the new region is combined with the previous region, selecting all of the text.

Return Value

The method will return the selected text. If the method fails because incorrect row or column values were used, or because an invalid option was specified, it will return an empty string.

Remarks

The **Select** method selects a region of the virtual display. This enables the application to select text in the same way that a user would by clicking and dragging the mouse over the display window. The **SelText** property can be used to return the text that has been selected by this method.

See Also

AutoSelect Property, SelText Property, Deselect 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 destroys the virtual display 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 virtual display.

Syntax

object.**Write**(*Buffer*, [*Length*])

Parameters

Buffer

A buffer variable that contains the data to be written to the display. If the variable is a **String** type, then then the data will be written as a string of characters. This is the most appropriate data type to use for text data that consists of printable characters. This method will also accept a **Byte** array of characters to be written to the virtual display.

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 characters to write is determined by the size of the buffer.

Return Value

This method returns the number of bytes actually written to the display, or -1 if an error was encountered.

Remarks

The **Write** method copies the data in ***Buffer*** to the virtual display at the current cursor location. If the data contains control characters or escape sequences, they will be processed according the **Emulation** property setting.

See Also

CodePage Property, Emulation Property, SelText Property Text Property

Terminal Emulator Control Events

Event	Description
Change	This event is generated when the contents of a control has changed
Click	This event is generated when the user presses and releases a mouse button
DblClick	This event is generated when the user presses and releases a mouse button twice
KeyDown	This event is generated when a key is pressed
KeyMapped	This event is generated when a mapped key is pressed
KeyPress	This event is generated when a key is pressed and released
KeyUp	This event is generated when a key is released
MouseDown	This event is generated when a mouse button is pressed
MouseMove	This event is generated when the user moves the mouse
MouseUp	This event is generated when a mouse button is released
Resize	The Resize event is generated after the control has been resized

Change Event

The **Change** event is generated when the contents of a control has changed.

Syntax

Sub *object*_**Change**([*Index* **As Integer**])

Remarks

The **Change** event is generated when the contents of a control have changed and can be used to synchronize or coordinate data display among controls. For example, you can use the control's **Change** event to check for text at certain screen location and then a list box which contains that text.

Modifying the contents of the virtual display by setting the **Text** property or calling the **Write** method in the event handler can cause a cascading event. It is recommended that you only read, not modify, the contents of the virtual display when this even occurs.

See Also

Resize Event

Click Event

The **Click** event is generated when the user presses and releases a mouse button.

Syntax

Sub *object*_**Click**([*Index* **As Integer**])

Remarks

The **Click** event is generated when the user presses and releases the mouse button anywhere over the control's window. To distinguish between the user pressing the left or right mouse buttons, use the **MouseDown** and **MouseUp** events. To determine the column and row in the virtual display where the mouse button was clicked, check the **MouseX** and **MouseY** properties.

When debugging events, do not use message boxes to show when the event occurred, as this will interfere with the normal functioning of many events. For example, displaying a message box in the **Click** event may prevent the **DblClick** event from being raised.

See Also

MouseX Property, MouseY Property, DblClick Event, MouseDown Event, MouseUp Event

DblClick Event

The **DblClick** event is generated when the user presses and releases a mouse button twice.

Syntax

Sub *object*_**DblClick**([*Index* **As Integer**])

Remarks

The **DblClick** event is generated when the user presses and releases the mouse button, then quickly presses and releases the mouse button again, anywhere over the control's window. To distinguish between the user pressing the left or right mouse buttons, use the **MouseDown** and **MouseUp** events. To determine the column and row in the virtual display where the mouse button was clicked, check the **MouseX** and **MouseY** properties.

If the user doesn't double-click within the system's time limit, the control receives another **Click** event instead. The double-click time limit may vary because the user can set the double-click speed in the Control Panel. When you're attaching procedures for these related events, be sure that their actions don't conflict. Controls that don't receive **DblClick** events may receive two clicks instead of a double-click.

When debugging events, do not use message boxes to show when the event occurred, as this will interfere with the normal functioning of many events. For example, displaying a message box in the **Click** event may prevent the **DblClick** event from being raised.

See Also

MouseX Property, MouseY Property, Click Event, MouseDown Event, MouseUp Event

KeyDown Event

The **KeyDown** event is generated when a key is pressed.

Syntax

Sub *object*_**KeyDown**([*Index* **As Integer**], *KeyCode* **As Integer**, *Shift* **As Integer**)

Remarks

The **KeyDown** event is generated when the user presses a key on the keyboard. The following arguments are passed to the event handler:

KeyCode

A key code which identifies the key being pressed. If the key is a number or letter on the keyboard, then the code corresponds to its ASCII equivalent. For example, the key code for the 'A' key is 65. It is important to note that shift states are handled differently in that the key code is the same regardless if the shift or control key is being pressed at the same time. For keys other than numbers or letters, consult the Virtual Key Constants table.

Shift

An integer that corresponds to the state of the Shift, Ctrl, and Alt keys at the time the key is pressed. The ***Shift*** argument is a bit field with the least-significant bits corresponding to the Shift key (bit 0), the Ctrl key (bit 1), and the Alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both Ctrl and Alt keys are pressed, the value of ***Shift*** is 6.

See Also

KeyMapped Event, KeyPress Event, KeyUp Event

KeyMapped Event

The **KeyMapped** event is generated when a mapped key is pressed.

Syntax

Sub *object*_**KeyMapped**([*Index* **As Integer**,] *KeyIndex* **As Integer**, *Shift* **As Integer**, *KeyString* **As String**)

Remarks

This event is generated when the user presses a special key while the emulation window has focus, and that key is mapped to a string using the KeyMap property array.

KeyIndex

An integer which specifies which mapped key was pressed, and is the same as the index value used with the **KeyMap** property array. Refer to the KeyMap Constants table for a list of keys and their corresponding values which may be mapped by the application. Note that the ***KeyIndex*** value is different than the key codes used by the **KeyDown** and **KeyUp** events.

Shift

An integer that corresponds to the state of the Shift, Ctrl, and Alt keys at the time the key is pressed. The ***Shift*** argument is a bit field with the least-significant bits corresponding to the Shift key (bit 0), the Ctrl key (bit 1), and the Alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both Ctrl and Alt keys are pressed, the value of ***Shift*** is 6.

KeyString

A string that contains the control and/or escape sequence that the key has been mapped to. This is the same value that was assigned to the **KeyMap** property array for this key.

Example

The following example demonstrates how to use the **KeyMapped** event in conjunction with the **SendKey** method in the Telnet control:

```
Private Sub Terminal1_KeyMapped(KeyIndex As Integer, Shift As Integer, KeyString As
String)
    TelnetClient1.SendKey KeyString
End Sub
```

See Also

KeyMap Property, KeyDown Event, KeyPress Event, KeyUp Event

KeyPress Event

The **KeyPress** event is generated when a key is pressed and released.

Syntax

Sub *object*_**KeyPress**([*Index* **As Integer**], *KeyAscii* **As Integer**)

Remarks

The **KeyPress** event is generated when the user presses and releases key on the keyboard. The following arguments are passed to the event handler:

KeyAscii

An integer which specifies the standard ASCII value of the key that was pressed. In Visual Basic, you can convert the key value into a character by using the **Chr** function.

Example

The following example demonstrates how to use the **KeyPress** event in conjunction with the **SendKey** method in the Telnet control:

```
Private Sub Terminal1_KeyPress(KeyAscii As Integer)
    TelnetClient1.SendKey KeyAscii
End Sub
```

See Also

KeyDown Event, KeyMapped Event, KeyUp Event

KeyUp Event

The **KeyUp** event is generated when a key is released.

Syntax

Sub *object*_**KeyUp**([*Index* **As Integer**], *KeyCode* **As Integer**, *Shift* **As Integer**)

Remarks

The **KeyUp** event is generated when the user releases a previously pressed key. The following arguments are passed to the event handler:

KeyCode

A key code which identifies the key being released. If the key is a number or letter on the keyboard, then the code corresponds to its ASCII equivalent. For example, the key code for the 'A' key is 65. It is important to note that shift states are handled differently in that the key code is the same regardless if the shift or control key is being pressed at the same time. For keys other than numbers or letters, consult the Virtual Key Constants table.

Shift

An integer that corresponds to the state of the Shift, Ctrl, and Alt keys at the time the key is released. The ***Shift*** argument is a bit field with the least-significant bits corresponding to the Shift key (bit 0), the Ctrl key (bit 1), and the Alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both Ctrl and Alt keys are pressed, the value of ***Shift*** is 6.

See Also

KeyDown Event, KeyMapped Event, KeyPress Event

MouseDown Event

The **MouseDown** event is generated when a mouse button is pressed.

Syntax

Sub *object*_**MouseDown**([*Index* **As Integer**], *Button* **As Integer**, *Shift* **As Integer**, *X* **As Integer**, *Y* **As Integer**)

Remarks

The **MouseDown** event is generated when the user presses a button on the mouse. The following arguments are passed to the event handler:

Button

Returns an integer that identifies the button that was pressed to cause the event. The ***Button*** argument is a bit field with bits corresponding to the left button (bit 0), right button (bit 1), and middle button (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Only one of the bits is set, indicating the button that caused the event.

Shift

An integer that corresponds to the state of the Shift, Ctrl, and Alt keys at the time the button is pressed. The ***Shift*** argument is a bit field with the least-significant bits corresponding to the Shift key (bit 0), the Ctrl key (bit 1), and the Alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both Ctrl and Alt keys are pressed, the value of ***Shift*** is 6.

X, Y

Returns the current location of the mouse pointer. The ***X*** and ***Y*** values are always expressed in terms of the coordinate system set by the container. To determine the current row and column where the mouse is positioned in the virtual display, use the **MouseX** and **MouseY** properties.

See Also

MouseX Property, MouseY Property, MouseMove Event, MouseUp Event

MouseMove Event

The **MouseMove** event is generated when the user moves the mouse.

Syntax

Sub *object*_**MouseMove**([*Index* **As Integer**], *Button* **As Integer**, *Shift* **As Integer**, *X* **As Integer**, *Y* **As Integer**)

Remarks

The **MouseMove** event is generated when the user moves the mouse over the control window. The following arguments are passed to the event handler:

Button

Returns an integer that corresponds to the state of the mouse buttons when the mouse was moved. The ***Button*** argument is a bit field with bits corresponding to the left button (bit 0), right button (bit 1), and middle button (bit 2). These bits correspond to the values 1, 2, and 4, respectively. It indicates the complete state of the mouse buttons; some, all, or none of these three bits can be set, indicating that some, all, or none of the buttons are pressed.

Shift

An integer that corresponds to the state of the Shift, Ctrl, and Alt keys at the time the mouse was moved. The ***Shift*** argument is a bit field with the least-significant bits corresponding to the Shift key (bit 0), the Ctrl key (bit 1), and the Alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both Ctrl and Alt keys are pressed, the value of ***Shift*** is 6.

X, Y

Returns the current location of the mouse pointer. The ***X*** and ***Y*** values are always expressed in terms of the coordinate system set by the container. To determine the current row and column where the mouse is positioned in the virtual display, use the **MouseX** and **MouseY** properties.

See Also

MouseX Property, MouseY Property, MouseDown Event, MouseUp Event

MouseUp Event

The **MouseUp** event is generated when a mouse button is released.

Syntax

Sub *object*_**MouseUp**([*Index* **As Integer**], *Button* **As Integer**, *Shift* **As Integer**, *X* **As Integer**, *Y* **As Integer**)

Remarks

The **MouseUp** event is generated when the user releases a previously pressed mouse button. The following arguments are passed to the event handler:

Button

Returns an integer that identifies the button that was released to cause the event. The ***Button*** argument is a bit field with bits corresponding to the left button (bit 0), right button (bit 1), and middle button (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Only one of the bits is set, indicating the button that caused the event.

Shift

An integer that corresponds to the state of the Shift, Ctrl, and Alt keys at the time the button is released. The ***Shift*** argument is a bit field with the least-significant bits corresponding to the Shift key (bit 0), the Ctrl key (bit 1), and the Alt key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both Ctrl and Alt keys are pressed, the value of ***Shift*** is 6.

X, Y

Returns the current location of the mouse pointer. The ***X*** and ***Y*** values are always expressed in terms of the coordinate system set by the container. To determine the current row and column where the mouse is positioned in the virtual display, use the **MouseX** and **MouseY** properties.

See Also

MouseX Property, MouseY Property, MouseDown Event, MouseMove Event

Resize Event

The Resize event is generated after the control has been resized.

Syntax

Sub *object*_**Resize** ([*Index* **As Integer**])

Remarks

This event can be used to move or resize other controls when the control is resized. You can also use the Resize event to recalculate the width and/or height of controls that depend on the size of the control window.

See Also

Change Event

Terminal Emulator Control Sequences

Terminal Control Sequences

<ESC>c	Reset display to initial state
<ESC>8	Display alignment test

Cursor Control Sequences

<ESC>D	Move cursor down to next line
<ESC>E	Move cursor to first column and down one line
<ESC>M	Move cursor up one line
<ESC>7	Save cursor position, attributes and colors
<ESC>8	Restore saved cursor position, attributes and colors
<ESC>[*n*A	Move cursor up *n* lines
<ESC>[*n*B	Move cursor down *n* lines
<ESC>[*n*C	Move cursor forward *n* spaces
<ESC>[*n*D	Move cursor backward *n* spaces
<ESC>[*n*E	Move cursor to beginning of line, down *n* lines
<ESC>[*n*F	Move cursor to beginning of line, up *n* lines
<ESC>[*x*G	Move cursor to column *x*
<ESC>[*y*;*x*H	Move cursor to line *y*, column *x*
<ESC>[*n*I	Move cursor forward *n* tabstops
<ESC>[*n*Z	Move cursor backwards *n* tabstops
<ESC>[*n*a	Move cursor forward *n* spaces
<ESC>[*y*d	Move cursor to row *y*
<ESC>[*n*e	Move cursor down *n* lines
<ESC>[*y*;*x*f	Move cursor to line *y*, column *x*
<ESC>[s	Save cursor position
<ESC>[u	Return to saved cursor position
<ESC>[x`	Move cursor to column *x*

Attribute and Color Sequence

Select display attributes and color

n Value	Description
0	Reset to default attributes and colors
1	Bold attribute
2	Dim attribute
4	Underline attribute
5	Blink attribute (same as reverse)
7	Reverse attribute
8	Hidden attribute
22	Clear bold attribute
24	Clear underline attribute
25	Clear blink attribute

<ESC>[*n*m	27	Clear reverse attribute
	29	Clear color attributes
	30	Black foreground
	31	Red foreground
	32	Green foreground
	33	Yellow foreground
	34	Blue foreground
	35	Magenta foreground
	36	Cyan foreground
	37	White foreground
	40	Black background
	41	Red background
	42	Green background
	43	Yellow background
	44	Blue background
	45	Magenta background
	46	Cyan background
	47	White background

Character Set Sequences

<ESC>(A	Assign ISO Latin 1 character set to font bank G0
<ESC>(B	Assign United States ASCII character set to font bank G0
<ESC>(0	Assign graphics character set to font bank G0
<ESC>)A	Assign ISO Latin 1 character set to font bank G1
<ESC>)B	Assign United States ASCII character set to font bank G1
<ESC>)0	Assign graphics character set to font bank G1

Erase Sequences

<ESC>[*n*@ Insert ***n*** blank spaces

Erase all or part of the display

<ESC>[*n*J

n Value	Description
0	From current position to end of display
1	From beginning of display to current position
2	Erase the entire display

Erase all or part of a line

<ESC>[*n*K

n Value	Description
0	From current position to end of line
1	From beginning of line to current position
2	Erase the entire line

<ESC>[*n*L Insert ***n*** new blank lines

<ESC>[*n*M	Delete *n* lines from current cursor position
<ESC>[*n*P	Delete *n* characters from current cursor position

Scrolling Sequences

<ESC>[*n*S	Scroll display up *n* lines
<ESC>[*n*T	Scroll display down *n* lines
<ESC>[*n*X	Erase *n* characters from the current position
<ESC>[*y1*;*y2*r	Set scrolling region from lines *y1* to *y2*

Keypad Sequences

<ESC>=	Place keypad into applications mode
<ESC>>	Place keypad into numeric mode

Emulation Option Sequences

<ESC>[?*n*h

Set emulation option

n Value	Description
1	Enable cursor key application mode
2	Enable ANSI escape sequences
5	Reverse foreground and background colors
6	Enable origin mode
7	Enable auto-wrap mode
20	Enable linefeed/newline mode
25	Display caret
66	Place keypad in applications mode

<ESC>[?*n*1

Set emulation option

n Value	Description
1	Disable cursor key application mode
2	Enable VT52 escape sequences
5	Restore foreground and background colors
6	Disable origin mode
7	Disable auto-wrap mode
20	Disable linefeed/newline mode
25	Hide caret
66	Place keypad in numeric mode

Console Escape Sequences

<ESC>[=*n*A	Set the overscan color (ignored)
<ESC>[=*n1*;*n2*B	Set bell sound (parameters ignored)
<ESC>[=*n1*;*n2*C	Set the caret size

<ESC>[=*n*D

Set background color intensity

n Value	Description
0	Decrease background color intensity

	1	Increase background color intensity
<ESC>[=*n*E	Set blink vs. bold attribute (ignored)	
<ESC>[=*n*F	Set normal foreground color	
<ESC>[=*n*G	Set normal background color	
<ESC>[=*n*H	Set reverse foreground color	
<ESC>[=*n*I	Set reverse background color	
<ESC>[=*n*J	Set graphics foreground color	
<ESC>[=*n*K	Set graphics background color	

n Value	Description
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	White
8	Gray
9	Light blue
10	Light green
11	Light cyan
12	Light red
13	Light magenta
14	Yellow
15	High White

Control Character Sequences

<CTL>G	Ring audible bell, if enabled
<CTL>H	Move cursor one character backwards
<CTL>I	Move cursor forward to next tabstop
<CTL>J	Move cursor down to next line
<CTL>M	Move cursor to beginning of line
<CTL>N	Select G1 character set
<CTL>O	Select G0 character set
<CTL>Z	Abort current escape sequence
<DEL>	Erase and move cursor one character backwards

KeyMap Constants

The following table lists the constants which can be used as the index value with the KeyMap property array to map control or escape sequences to certain keys. These values are also used by the KeyMapped event when the user presses a mapped key.

Value	Description
nvtF1	F1 function key
nvtF2	F2 function key
nvtF3	F3 function key
nvtF4	F4 function key
nvtF5	F5 function key
nvtF6	F6 function key
nvtF7	F7 function key
nvtF8	F8 function key
nvtF9	F9 function key
nvtF10	F10 function key
nvtF11	F11 function key
nvtF12	F12 function key
nvtShiftF1	Shift F1 function key
nvtShiftF2	Shift F2 function key
nvtShiftF3	Shift F3 function key
nvtShiftF4	Shift F4 function key
nvtShiftF5	Shift F5 function key
nvtShiftF6	Shift F6 function key
nvtShiftF7	Shift F7 function key
nvtShiftF8	Shift F8 function key
nvtShiftF9	Shift F9 function key
nvtShiftF10	Shift F10 function key
nvtShiftF11	Shift F11 function key
nvtShiftF12	Shift F12 function key
nvtEnter	Enter key
nvtErase	Backspace key

Value	Description
nvtUp	Cursor up key
nvtDown	Cursor down key
nvtLeft	Cursor left key
nvtRight	Cursor right key
nvtInsert	Insert key
nvtDelete	Delete key
nvtHome	Home key
nvtEnd	End key
nvtPageUp	Page up key
nvtPageDown	Page down key
nvtArrowUp	Up arrow key
nvtArrowDown	Down arrow key
nvtArrowLeft	Left arrow key
nvtArrowRight	Right arrow key
nvtKeypadEnter	Keypad enter key
nvtKeypad0	Numeric keypad 0
nvtKeypad1	Numeric keypad 1
nvtKeypad2	Numeric keypad 2
nvtKeypad3	Numeric keypad 3
nvtKeypad4	Numeric keypad 4
nvtKeypad5	Numeric keypad 5
nvtKeypad6	Numeric keypad 6
nvtKeypad7	Numeric keypad 7
nvtKeypad8	Numeric keypad 8
nvtKeypad9	Numeric keypad 9

Virtual Key Constants

The following table lists the virtual key codes which are used by the KeyDown, KeyMapped and KeyUp events. The constants listed are those that are defined by Visual Basic. If you are using another programming language, consult your technical reference documentation for more information about the constants used to define virtual key codes.

Value	Description
vbKeyLButton	Left mouse button
vbKeyRButton	Right mouse button
vbKeyMButton	Middle mouse button
vbKeyBack	Backspace key
vbKeyTab	Tab key
vbKeyReturn	Return or enter key
vbKeyShift	Shift key
vbKeyControl	Control key
vbKeyPause	Pause/Break key
vbKeyCaptial	Caps Lock key
vbKeyEscape	Escape key
vbKeySpace	Space key
vbKeyPageUp	Page Up key
vbKeyPageDown	Page Down key
vbKeyEnd	End key
vbKeyHome	Home key
vbKeyLeft	Left arrow key
vbKeyUp	Up arrow key
vbKeyRight	Right arrow key
vbKeyDown	Down arrow key
vbKeyPrint	Print key
vbKeyInsert	Insert key
vbKeyDelete	Delete key
vbKeyHelp	Help key
vbKeyNumpad0	Number pad 0 key
vbKeyNumpad1	Number pad 1 key
vbKeyNumpad2	Number pad 2 key
vbKeyNumpad3	Number pad 3 key

Value	Description
vbKeyNumpad5	Number pad 5 key
vbKeyNumpad6	Number pad 6 key
vbKeyNumpad7	Number pad 7 key
vbKeyNumpad8	Number pad 8 key
vbKeyNumpad9	Number pad 9 key
vbKeyMultiply	Number pad * key
vbKeyAdd	Number pad + key
vbKeySubtract	Number pad - key
vbKeyDecimal	Number pad decimal key
vbKeyDivide	Number pad / key
vbKeyF1	F1 key
vbKeyF2	F2 key
vbKeyF3	F3 key
vbKeyF4	F4 key
vbKeyF5	F5 key
vbKeyF6	F6 key
vbKeyF7	F7 key
vbKeyF8	F8 key
vbKeyF9	F9 key
vbKeyF10	F10 key
vbKeyF11	F11 key
vbKeyF12	F12 key
vbKeyF13	F13 key
vbKeyF14	F14 key
vbKeyF15	F15 key
vbKeyF16	F16 key
vbKeyNumlock	Num Lock key
vbKeyScrollLock	Scroll Lock key

vbKeyNumpad4	Number pad 4 key			

Text Message Control

Send text messages to a mobile communications device using a gateway service.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	TextMessageCtl.TextMessage
File Name	CSTXTX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.TextMessage.11
ClassID	B39C74C3-F9CB-487A-A1B0-E69F7FF807A2
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None

Overview

Short Message Service (SMS) is a text messaging service used by mobile communication devices to exchange brief text messages. Most service providers also provide gateway servers that can be used to send messages to a wireless device on their network using standard email protocols. The **TextMessage** control provides methods that can be used to determine the provider associated with a specific telephone number and send a text message to the device using the provider's mail gateway.

This control has been designed to assist developers in sending text message notifications as part of their application. For example, it can be used to enable your software to automatically send notifications when a specific event occurs, such as an error condition. This control is not designed to be used with software that will send out a large number of text messages to many users, and there are limitations on the number of messages that may be sent to different phone numbers over a short period of time. Because many recipients must pay a fee for each text message they receive, text messages should only be sent to those who explicitly request them.

Note: This component only supports service providers in North America and cannot be used to send text messages to mobile devices that use providers outside of the United States and Canada.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop

and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

Text Message Control Properties

Property	Description
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 to occur
Message	Gets and sets the current message text
Password	Gets and sets the password used to authenticate the session
PhoneNumber	Gets and sets the phone number for the mobile device
Provider	Gets and sets the name of the preferred wireless service provider
ProviderCount	Gets the number of supported wireless service providers
ProviderName	Gets the name of a supported wireless service provider
Relay	Enable or disable the use of an intermediate relay server to send messages
Secure	Enable or disable secure connections to the server
Sender	Gets and sets the value that identifies the sender of the message
ServerName	Gets and sets the name of the server that is used to send messages
ServerPort	Gets and sets the port number used to establish a connection
ServiceType	Gets and sets the type of service used to send messages
ThrowError	Enable or disable error handling by the container of the control
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
Urgent	Enable or disable the option that flags a message as urgent
UserName	Gets and sets the user name or ID used to authenticate the session
Version	Gets the current version of the object

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 to occur

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, ThrowError Property, OnError Event

Message Property

Gets and sets the current message text.

Syntax

object.**Message** [= *value*]

Remarks

The **Message** property returns the current message text. Changing the value of this property will change the message text sent by the **SendMessage** method if no message is explicitly specified. In most cases, a message should not exceed 160 characters in length, although some service providers may accept longer messages. If a message exceeds the maximum number of characters accepted by a service provider, the message may be ignored or it may be split into multiple messages.

Data Type

String

See Also

PhoneNumber Property, Sender Property, Urgent Property, SendMessage Method

Password Property

Gets and sets the password used to authenticate the session.

Syntax

object.**Password** [= *value*]

Remarks

The **Password** property is used to authenticate the current session. If the message is being sent using SMTP, this property is used to specify a password when connecting to the mail server . If no authentication is required, this property may be an empty string. Note that some service providers may use terminology other than "password" with their documentation; in that case, this property will always specify the second of a pair of authentication tokens sent to the server to identify the client.

Data Type

String

See Also

ServerName Property, ServerPort Property, UserName Property, SendMessage Method

PhoneNumber Property

Gets and sets the phone number for the mobile device.

Syntax

object.**PhoneNumber** [= *value*]

Remarks

The **PhoneNumber** property returns the current phone number. Changing the value of this property will change the default phone number the **SendMessage** method will use when sending a message. This can be a standard E.164 formatted phone number or an unformatted number. Any extraneous whitespace, punctuation or other non-numeric characters in the string will be ignored.

Data Type

String

See Also

Message Property, Provider Property, Sender Property, SendMessage Method

Provider Property

Gets and sets the name of the preferred wireless service provider.

Syntax

object.**Provider** [= *value*]

Remarks

The **Provider** property returns the preferred wireless service provider associated with the current phone number. Changing the value of this property will change the preferred wireless service provider. If this property is an empty string, the default provider assigned to the recipient's phone number will be used. This property is only used with messages sent using SMTP and is ignored for other message services.

In the United States and Canada, most wireless common carriers are required to provide wireless number portability (WNP) which allows a customer to continue to use their current phone number even if they switch to another service provider. This can result in a situation where a specific phone number is shown as allocated to one provider, but in actuality that user has switched to a different provider. For example, a user may have originally purchased a phone and service with AT&T and then later switched to Verizon, but decided to keep their phone number. In this case, if Verizon was not specified as the preferred provider, the library would attempt to send the message to the AT&T gateway, since that was the original provider who allocated the phone number.

For most applications, the correct way to handle the situation in which a user may have switched to a different service provider is to allow them to select a preferred provider in your user interface. For example, you could display a drop-down list of available providers for them to select from, populated using the **ProviderName** property array. If they select a preferred provider, then you would assign that value to this property. If they choose to use the default provider, set this property to an empty string.

Data Type

String

See Also

ProviderCount Property, ProviderName Property, GetAddress Method, GetProvider Method

ProviderCount Property

Gets the number of supported wireless service providers.

Syntax

object.**ProviderCount**

Remarks

The **ProviderCount** property returns the number of wireless service providers supported by the control. This property is used in conjunction with the **ProviderName** property to enumerate all of the supported service providers. Typically this done to populate a user-interface control that enables the user to select a preferred service provider.

Data Type

Integer (Int32)

Example

```
With TextMessage1
    For nIndex = 0 To .ProviderCount - 1
        ComboBox1.AddItem .ProviderName(nIndex)
    Next
End With
```

See Also

Provider Property, ProviderName Property, GetAddress Method, GetProvider Method

ProviderName Property

Gets the name of a supported wireless service provider.

Syntax

object.**ProviderName**(*index*)

Remarks

The **ProviderName** property array returns the name of supported wireless service provider. This property is used in conjunction with the **ProviderCount** property to enumerate all of the supported service providers. Typically this done to populate a user-interface control that enables the user to select a preferred service provider. The first provider name has an index value of zero. Specifying an invalid index value will cause the control to throw an exception.

Data Type

String

Example

```
With TextMessage1
    For nIndex = 0 To .ProviderCount - 1
        ComboBox1.AddItem .ProviderName(nIndex)
    Next
End With
```

See Also

Provider Property, ProviderCount Property, GetAddress Method, GetProvider Method

Relay Property

Enable or disable the use of an intermediate relay server to send messages.

Syntax

object.**Relay** [= { True | False }]

Remarks

The **Relay** property is used to determine if the control will send the message directly to the wireless service provider's gateway server, or if the message will be relayed through another mail server. The default value for this property is False. Setting this property to True will cause the control to connect to the mail server specified by the **ServerName** property, authenticate the client session if necessary and then submit the message.

When a text message is sent using the SMTP service, the default action is to attempt to connect directly to the wireless service provider's gateway server. However, many residential Internet service providers (ISPs) do not permit their customers to connect to third-party mail servers and will block the outbound connection. Some wireless service providers may also reject messages that originate from residential IP addresses.

To resolve this issue, the developer should allow the user to specify an alternate mail server that will relay the message to the wireless service provider. For residential users, this will typically be the mail server provided by their ISP. For business users, this will usually be their corporate mail server. The **ServerName** and **ServerPort** properties are used to identify the relay server, and the **UserName** and **Password** properties provide the credentials to authenticate the client session.

Data Type

Boolean

See Also

Password Property, ServerName Property, ServerPort Property, UserName Property

Secure Property

Enable or disable secure connections to the server.

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 must set this property value to True prior to calling the **SendMessage** method.

This property is only used when sending a message through a relay server using the SMTP service. Messages that are sent directly to the wireless service provider's gateway do not use authentication and are not secure, regardless of the value of this property.

Data Type

Boolean

See Also

Relay Property, ServerName Property, ServerPort Property, SendMessage Method

Sender Property

Gets and sets the value that identifies the sender of the message.

Syntax

object.**Sender** [= *value*]

Remarks

The **Sender** property returns the email address of the sender when the SMTP service is used to send the message. For other service types, this property typically specifies the phone number or shortcode associated with the sender. Assigning a value to this property will set the default sender when the **SendMessage** method is called.

Data Type

String

See Also

Message Property, PhoneNumber Property

ServerName Property

Gets and sets the name of the server that is used to send messages.

Syntax

object.**ServerName** [= *value*]

Remarks

The **ServerName** property is used to specify an alternate server that is used to deliver messages. When the SMTP service is used, this property is used in conjunction with the **Relay** property to enable the relaying of messages through another mail server.

By default, the phone number is used to automatically determine the host name of the gateway server that is responsible for accepting messages for the mobile device. However, under some circumstances it may not be possible to send messages directly to the wireless service provider's gateway. For example, many Internet service providers (ISPs) require that customers relay all messages through their servers and block any attempt to establish a direct connection with another mail server. This property can be used to specify an alternate server that will be responsible for sending the message.

Data Type

String

See Also

Relay Property, ServerName Property, ServerPort Property

ServerPort Property

Gets and sets the port number used to establish a connection.

Syntax

object.**ServerPort** [= *value*]

Remarks

The **ServerPort** property defines the port number which is used to establish a connection with the server. This property is used in conjunction with the **ServerName** property to specify an alternate server which is responsible for delivering messages.

Data Type

Integer (Int32)

See Also

Relay Property, ServerName Property

ServiceType Property

Gets and sets the type of service used to send messages.

Syntax

object.**ServiceType** [= *value*]

Remarks

The **ServiceType** property identifies the service used to send the text message. This property is provided for future expansion where different types of services may be used to send a message. Currently, the only service type that is supported is SMTP, where the message is sent through a gateway mail server.

The following values may be assigned to or returned by this property:

Value	Description	
1	smsServiceSmtp	The text message will be sent through the mail gateway for the specified service provider. This service uses SMTP to submit the message for delivery, either directly to the server provider's mail gateway server or through a relay server. This is the default service type.

Data Type

Integer (Int32)

See Also

Message Property, PhoneNumber Property, Sender Property, SendMessage Method

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** [= { True | False }]

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
TextMessage1.ThrowError = False
nError = TextMessage1.SendMessage()

If nError > 0 Then
    MsgBox TextMessage1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

TextMessage1.ThrowError = True
TextMessage1.SendMessage

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error. This includes the amount of time the control will spend attempting to connect to a server and if the connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

See Also

SendMessage Method

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the TextMessage and NewsFeed controls, and you set the **Trace** property to True on the TextMessage control, function calls made by both controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
controlTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
controlTraceError	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.
controlTraceWarning	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.
controlTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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

Enable or disable the option that flags a message as urgent.

Syntax

object.**Urgent** [= { True | False }]

Remarks

The **Urgent** property specifies whether a message will be flagged as urgent or not. If this property is set to True, the message will sent with a high priority. Note that this does not guarantee the message will be received any differently than a standard text message. Each wireless service provider may handle urgent messages differently, and some providers may simply ignore the message priority. By default, this property is False.

Data Type

Boolean

See Also

Message Property, SendMessage Method

UserName Property

Gets and sets the user name or ID used to authenticate the session.

Syntax

object.**UserName** [= *value*]

Remarks

The **UserName** property is used to authenticate the current session. If the message is being sent using SMTP, this property is used to specify a user name when connecting to the mail server . If no authentication is required, this property may be an empty string. Note that some service providers may use terminology other than "username" with their documentation; in that case, this property will always specify the first of a pair of authentication tokens sent to the server to identify the client.

Data Type

String

See Also

Password Property, ServerName Property, ServerPort Property, SendMessage Method

Version Property

Return the current version of the object.

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

Text Message Control Methods

Method	Description
GetAddress	Get the email address associated with the specified phone number
GetProvider	Get the name of the wireless service provider associated with a phone number
Initialize	Initialize the control and validate the runtime license key
Reset	Reset the internal state of the control
SendMessage	Send a text message to the specified mobile device
Uninitialize	Uninitialize the control and release any system resources that were allocated

GetAddress Method

Get the email address associated with the specified phone number.

Syntax

object.**GetAddress**(*PhoneNumber*, *Address*)

Parameters

PhoneNumber

A string value which specifies the phone number of the mobile device. This can be a standard E.164 formatted number or an unformatted number. Any extraneous whitespace, punctuation or other non-numeric characters in the string will be ignored.

Address

A string that will contain the email address associated with the specified phone number when the method returns. A mail message sent to this address will be forwarded to the mobile device as a text message. This parameter must be passed by reference.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure. If the method fails, the value of the **LastError** property can be used to determine cause of the failure.

Remarks

The **GetAddress** method can be used to determine if a phone number is associated with a mobile device and obtain the email address for the wireless service provider's gateway. This is done by sending an query to a server that will check the phone number against a database of known providers and the phone numbers that have been allocated for wireless devices.

By default, this method will attempt to automatically determine which service provider is associated with the phone number. If the service provider cannot be determined, this method will fail and return an error code. If the **Provider** property has been set to to the name of a specific service provider, that provider will be used to determine the gateway email address.

This method sends an HTTP query to the server **api.sockettools.com** to obtain information about the phone number and wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the method will fail and an appropriate error will be returned. The server imposes a limit on the maximum number of connections that can be established and the maximum number of requests that can be issued per minute. If this method is called multiple times over a short period, the control may also force the application to block briefly. Server responses are cached per session, so calling this method multiple times using the same phone number will not increase the request count.

See Also

Provider Property, ProviderCount Property, ProviderName Property, GetProvider Method, SendMessage Method

GetProvider Method

Get the name of the wireless service provider associated with a phone number.

Syntax

object.**GetProvider**(*PhoneNumber*, *ProviderName*)

Parameters

PhoneNumber

A string value which specifies the phone number of the mobile device. This can be a standard E.164 formatted number or an unformatted number. Any extraneous whitespace, punctuation or other non-numeric characters in the string will be ignored.

ProviderName

A string that will contain the name of the wireless service provider associated with the specified phone number when the method returns. This parameter must be passed by reference.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure. If the method fails, the value of the **LastError** property can be used to determine cause of the failure.

Remarks

The **GetProvider** method can be used to determine if a phone number is associated with a mobile device and obtain the name of the service provider associated with the number. This is done by sending an query to a server that will check the phone number against a database of known providers and the phone numbers that have been allocated for wireless devices.

This method sends an HTTP query to the server **api.sockettools.com** to obtain information about the phone number and wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the method will fail and an appropriate error will be returned. The server imposes a limit on the maximum number of connections that can be established and the maximum number of requests that can be issued per minute. If this method is called multiple times over a short period, the control may also force the application to block briefly. Server responses are cached per session, so calling this method multiple times using the same phone number will not increase the request count.

See Also

Provider Property, ProviderCount Property, ProviderName Property, GetAddress Method, SendMessage Method

Initialize Method

Initialize the control and validate the runtime license key.

Syntax

object.**Initialize**([*LicenseKey*])

Parameters

LicenseKey

An optional string 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 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 smsClient = CreateObject("SocketTools.TextMessage.11")

nError = smsClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize 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, open network connections will be closed and any handles allocated by the control will be released.

See Also

Initialize Method, Uninitialize Method

SendMessage Method

Send a text message to the specified mobile device.

Syntax

object.**SendMessage**([*PhoneNumber*,] [*Sender*,] [*Message*,] [*Options*])

Parameters

PhoneNumber

An optional string value which specifies the phone number of the mobile device. This can be a standard E.164 formatted number or an unformatted number. Any extraneous whitespace, punctuation or other non-numeric characters in the string will be ignored. If this parameter is omitted, the current value of the **PhoneNumber** property will be used.

Sender

An optional string value that identifies the sender of the message. For messages being sent using SMTP, this should be a valid email address. If this parameter is omitted, the current value of the **Sender** property will be used.

Message

An optional string value that contains the text message that will be sent. If this parameter is omitted, the current value of the **Message** property will be used.

Options

An optional integer value that is reserved for future use. If it is specified, it should have a value of zero.

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure. If the method fails, the value of the **LastError** property can be used to determine cause of the failure.

Remarks

The **SendMessage** method is used to send a text message to the specified mobile device. This control is designed to support multiple methods of sending text messages, with the **ServiceType** property determining how the message is sent:

smsServiceSmtp

This message service sends the message through the wireless service provider's mail gateway using the SMTP protocol. However, it is important to note that many of these gateways will not accept messages from a client that is connected to them using a residential Internet service provider. If the application is being run on a system that uses a residential provider, that service provider may also block outbound connections to all mail servers other than their own. These anti-spam measures typically require that most end-user applications specify a relay mail server rather than submitting the message directly to the wireless provider's gateway.

By default, this method will attempt to automatically determine which service provider is associated with the phone number. If the service provider cannot be determined, this method will fail and return an error code. If the **Provider** property has been set to to the name of a specific service provider, that provider will be used to determine the gateway email address.

Because most wireless carriers in the United States and Canada must provide for wireless number portability, there is the possibility that the provider information returned may no longer correspond to

the telephone number. It is recommended that you provide your end-user with the ability to specify an alternate preferred provider to use when sending the text message. For more information, refer to **ProviderCount** and **ProviderName** properties.

This method sends an HTTP query to the server **api.sockettools.com** to obtain information about the phone number and wireless service provider. This requires that the local system can establish a standard network connection over port 80. If the client cannot connect to the server, the method will fail and an appropriate error will be returned. The server imposes a limit on the maximum number of connections that can be established and the maximum number of requests that can be issued per minute. If this method is called multiple times over a short period, the control may also force the application to block briefly. Server responses are cached per session, so calling this method multiple times using the same phone number will not increase the request count.

See Also

Message Property, PhoneNumber Property, Provider Property, Sender Property, ServiceType Property

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

Text Message Control Events

Event	Description
OnError	This event is generated when a control error occurs

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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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

Time Protocol Control

Query a time server for the current time and synchronize the local system clock with that value.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	TimeClientCtl.TimeClient
File Name	CSTIMX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.TimeClient.11
ClassID	F23074C6-02B3-4B88-8A8B-8AD2C6FB52F0
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 868

Overview

The Time Protocol control provides an interface for synchronizing the local system's time and date with that of a server. The time values returned are in in Coordinated Universal Time and be adjusted for the local host's timezone. The control enables developers to query a server for the current time and then update the system clock if desired.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

Time Protocol Control Properties

Property	Description
AutoResolve	Determines if host names and IP addresses are automatically resolved
HostAddress	Gets and sets the IP address of the server
HostName	Gets and sets the name of the server
Interval	Gets and sets the timer interval in milliseconds
IsBlocked	Return if the control is blocked performing an operation
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 to occur
LocalDate	Returns the network date adjusted for the local timezone
LocalTime	Returns the network time adjusted for the local timezone
RemotePort	Gets and sets the port number for a remote connection
SystemDate	Returns the network date adjusted for the local timezone
SystemTime	Returns the network time adjusted for the local timezone
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 object

AutoResolve Property

Determines if host names and IP 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 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 **AutoResolve** to False.

Data Type

Boolean

See Also

HostAddress Property, HostName Property

HostAddress Property

Gets and sets the IP address of the server.

Syntax

object.**HostAddress** [= *ipaddress*]

Remarks

The **HostAddress** property can be used to set the IP address for a server 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, HostName Property

HostName Property

Gets and sets the name of the server.

Syntax

object.**HostName** [= *hostname*]

Remarks

The **HostName** property should be set to the name of the server 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

Interval Property

Gets and sets the timer interval in milliseconds.

Syntax

object.**Interval** [= *milliseconds*]

Remarks

The **Interval** property specifies the number of milliseconds between calls to the **Timer** 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

Timer 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.

Note that this property will return True if there is ***any*** blocking operation being performed by the application, regardless if the specified control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

LastError Property

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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

LocalDate Property

Returns the network date adjusted for the local timezone.

Syntax

object.**LocalDate**

Remarks

The **LocalDate** property returns the network date and adjusts the value for the local timezone. The date is returned as a string formatted using the Short Date format for the current locale.

Data Type

String

See Also

LocalTime Property, SystemDate Property, SystemTime Property, GetTime Method, SetTime Method

LocalTime Property

Returns the network time adjusted for the local timezone.

Syntax

object.**LocalTime**

Remarks

The **LocalTime** property returns the network time and adjusts the value for the local timezone. The time is returned as a string formatted using the standard format for the current locale.

Data Type

String

See Also

LocalDate Property, SystemDate Property, SystemTime Property, GetTime Method, SetTime Method

RemotePort Property

Gets and sets the port number for a remote connection.

Syntax

object.**RemotePort** [= *portnumber*]

Remarks

The **RemotePort** property is used to set the port number that the control will use to establish a connection with the server.

Data Type

Integer (Int32)

See Also

HostAddress Property, HostName Property

SystemDate Property

Returns the network date adjusted for the local timezone.

Syntax

object.**SystemDate**

Remarks

The **SystemDate** property returns the network date in Coordinated Universal Time (UTC). The date is returned as a string formatted using the Short Date format for the current locale.

Data Type

String

See Also

LocalDate Property, LocalTime Property, SystemTime Property, GetTime Method

SystemTime Property

Returns the network time adjusted for the local timezone.

Syntax

object.**LocalTime**

Remarks

The **SystemTime** property returns the network time in Coordinated Universal Time (UTC). The time is returned as a string formatted using the standard format for the current locale.

Data Type

String

See Also

LocalDate Property, LocalTime Property, SystemDate Property, GetTime Method, SetTime Method

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
TimeClient1.ThrowError = False
strValue = TimeClient1.GetTime(strHostName)

If Len(strValue) = 0 Then
    MsgBox TimeClient1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

TimeClient1.ThrowError = True
strValue = TimeClient1.GetTime(strHostName)

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error.

Note that the **Timeout** property also determines the amount of time the control will spend attempting to connect to a server. If a connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
timeTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
timeTraceError	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.
timeTraceWarning	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.
timeTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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 object.

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

Time Protocol Control Methods

Method	Description
Cancel	Cancels the current blocking network operation
GetTime	Obtain the current system date and time
Initialize	Initialize the control and validate the runtime license key
Reset	Reset the internal state of the control
SetTime	Set the local system clock to the value returned by the network time server
Uninitialize	Uninitialize the control and release any system resources that were allocated

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

GetTime Method

Obtain the current system date and time.

Syntax

object.**GetTime**([*RemoteHost*], [*RemotePort*], [*Timeout*], [*Localize*])

Parameters

RemoteHost

A string which specifies the host name or IP address of the server. If this argument is not specified, it defaults to the value of the **HostAddress** property if it is defined. Otherwise, it defaults to the value of the **HostName** property.

RemotePort

A number which specifies the port to connect to on the server. If this argument is not specified, it defaults to the value of the **RemotePort** property. A value of zero indicates that the default port number for this service should be used to establish the connection.

Timeout

The number of seconds that the client will wait for a response before failing the operation. If this argument is not specified, the value of the **Timeout** property will be used as the default.

Localize

A boolean argument which specifies if the time should be localized to the timezone on the current system. If the value is True, then the time returned by the server will be adjusted so that it reflects the current time in the local timezone. If this argument is omitted or a value of false is passed to the method, the date and time are returned in Coordinated Universal Time (UTC).

Return Value

The time and date retrieved from the server will be returned as a string formatted according to the user's current locale. If the date could not be retrieved, an empty string will be returned. To determine the cause of the failure, check the value of the **LastError** property.

Remarks

The **GetTime** method causes the control to connect to the specified server and request the current date and time. In the United States, the National Institute of Standards and Technology (NIST) hosts a number of public servers which can be used to obtain the current time. The following table lists the current host names and addresses:

Server Name	IP Address	Location
time-a.nist.gov	129.6.15.28	Gaithersburg, Maryland
time-b.nist.gov	129.6.15.29	Gaithersburg, Maryland
time-nw.nist.gov	131.107.13.100	Redmond, Washington
time-a.timefreq.bldrdoc.gov	132.163.4.101	Boulder, Colorado
time-b.timefreq.bldrdoc.gov	132.163.4.102	Boulder, Colorado
time-c.timefreq.bldrdoc.gov	132.163.4.103	Boulder, Colorado

Time servers are also commonly maintained by Internet service providers and universities. If you are unable to obtain the time from a server, contact the system administrator to determine if they have

the standard time service available on port 37.

See Also

LocalDate Property, LocalTime Property, SystemDate Property, SystemTime Property, SetTime 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 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 timeClient = CreateObject("SocketTools.TimeClient.11")

nError = timeClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize 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, open network connections will be closed and any handles allocated by the control will be released.

See Also

Cancel Method, Initialize Method, Uninitialize Method

SetTime Method

Set the local system clock to the value returned by the network time server.

Syntax

object.**SetTime**([*NetworkTime*])

Parameters

NetworkTime

An optional date value which specifies the new system time. If this is passed as a string, it will be parsed according to the local system settings; if the string is not in a recognizable format, the method will return an error. Note that the string must specify the full date and time, not simply a time value. If this argument is omitted, the date and time value last returned from a call to the **GetTime** method will be used. This enables an application to synchronize the local system time with the time returned by the server.

Return Value

A boolean value of true is returned if the operation was successful, otherwise the method will return false. Check the value of the **LastError** property to determine the cause of the failure.

Remarks

The **SetTime** method causes the control to update the local system's clock to the specified date and time. It is required that the user have the appropriate privileges required to change the system clock, otherwise an error will be returned.

See Also

LocalDate Property, LocalTime Property, SystemDate Property, SystemTime Property, GetTime 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. 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

Time Protocol Control Events

Event	Description
OnCancel	This event is generated when a blocking operation is canceled
OnError	This event is generated when a control error occurs
OnTimeout	This event is generated when a blocking operation times out
OnTimer	This event is generated when the control's preset timer interval expires

OnCancel Event

The **OnCancel** event is generated when a blocking operation is canceled.

Syntax

Sub *object*_**OnCancel** ([*Index* **As Integer**])

Remarks

This event is generated 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, OnTimeout 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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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

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

The **OnTimer** is generated when the control's timer interval has elapsed. The frequency is specified in milliseconds by setting the **Interval** property.

See Also

Interval Property

Web Location Control

Get physical location information for the local computer system.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	WebLocationCtl.WebLocation
File Name	CSWIPX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.WebLocation.11
ClassID	D4356555-4D0E-4EB7-B280-25464D1CA23A
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None

Overview

The Web Location control returns information about the location associated with the with the external IP address of the local system. The accuracy of this information can vary depending on the location, with the most detailed information being available for North America. The country and time zone information for all locations is generally accurate. However, as the location information becomes more precise, details such as city names, postal codes and specific geographic locations (e.g.: longitude and latitude) may have reduced accuracy.

This location information should not be used by programs that require extremely accurate map coordinates, such as navigation applications. The location information in North America should be generally accurate within a 25 mile (40km) radius. However, given the nature of how IP address location works, there is no guarantee that location information for any specific IP address or network will be accurate.

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

This control uses SocketTools Web Services and will only function if there is an active Internet connection and the local system is capable of establishing a secure connection to the location service.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control

specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

Web Location Control Properties

Property	Description
ASNumber	Gets the autonomous system number for the network
CityName	Gets the city name for the current location
Coordinates	Gets the GPS coordinates for the current location
CountryAlpha	Gets the ISO alpha-2 code for the country at the current location
CountryCode	Gets the numeric code for the country at the current location
CountryName	Gets the name of the country for the current location
IPAddress	Gets the external IP address for the local system
IsInitialized	Determine if the control has been initialized
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Returns a description of the last error to occur
LastUpdate	Gets the number of seconds since the last location update
Latitude	Gets the latitude for the current location
LocalTime	Gets the local time adjusted for the location's time zone
LocationId	Gets a unique identifier for the current location
Longitude	Gets the longitude for the current location
Organization	Gets the name of the organization that owns the network
PostalCode	Gets the postal code for the current location
RegionCode	Gets the numeric region code for the current location
RegionName	Gets the region name for the current location
Subdivision	Gets the subdivision name for the current location
SubdivisionCode	Gets the alphanumeric code for the current subdivision
ThrowError	Enable or disable error handling by the container of the control
Timeout	Gets and sets the amount of time until a blocking operation fails
Timezone	Gets the time zone for the current location
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
TzOffset	Gets the time zone offset in seconds for the current location
TzShortName	Gets an abbreviated time zone name for the current location
Version	Return the current version of the object

ASNumber Property

Gets the autonomous system number for the network.

Syntax

object.**ASNumber**

Remarks

This property returns an integer value which is used to uniquely identify a global network (autonomous system) that is connected to the Internet. This number is assigned by regional registries and used by large networks, such as Internet Service Providers, for exchanging routing information with one another. This value can be used to determine the ownership of a particular network.

Data Type

Integer (Int32)

See Also

IPAddress Property, LocationId Property, Organization Property

CityName Property

Gets the city name for the current location.

Syntax

object.**CityName**

Remarks

This property returns a string which identifies the city in which the external IP address is located. These names will always be in English, regardless of the current system locale. If the city name cannot be determined, this property may return an empty string.

Data Type

String

See Also

CountryName Property, PostalCode Property, RegionName Property, Subdivision Property

Coordinates Property

Gets the GPS coordinates for the current location.

Syntax

object.**Coordinates**

Remarks

This property returns a string which specifies the location expressed using the Universal Transverse Mercator (UTM) coordinate system with the WGS-84 ellipsoid. UTM coordinates are commonly used with the Global Positioning System (GPS) and are comprised of three parts: the zone, the easting (the eastward-measured distance or *x*-coordinate) and the northing (the northward-measured distance or *y*-coordinate). An example of a string value returned by this property would be "14S 702089E 3646476N".

Data Type

String

See Also

Latitude Property, Longitude Property

CountryAlpha Property

Gets the ISO alpha-2 code for the country at the current location.

Syntax

object.**CountryAlpha**

Remarks

This property returns a string which contains the ISO 3166-1 alpha-2 code for the country the external IP address is located in. This is a two-letter country code established by the International Organization for Standardization (ISO).

The **SubdivisionCode** property can be used to determine the standard ISO standard code for a regional subdivision (such as a state, province or territory). For example, if the IP address is located in Los Angeles, California the **CountryAlpha** property would return "US" and the **SubdivisionCode** property would return "CA".

Data Type

String

See Also

CountryCode Property, CountryName Property, RegionName Property, Subdivision Property, SubdivisionCode Property

CountryCode Property

Gets the numeric code for the country at the current location.

Syntax

object.**CountryCode**

Remarks

This property returns an integer value which identifies the country where the external IP address is located. These codes can be up to three digits (usually displayed with leading zeros as necessary) and correspond to the country codes assigned by the United Nations. For example, the code for the United States is 840. It is important to note that these are not international dialing codes and should not be used with telephony applications.

Data Type

Integer (Int32)

See Also

CountryAlpha Property, CountryName Property, RegionName Property, Subdivision Property

CountryName Property

Gets the name of the country for the current location.

Syntax

object.**CountryName**

Remarks

This property returns a string which contains the full name of the country in which the external IP address is located. These names will always be in English, regardless of the current system locale.

Data Type

String

See Also

CountryAlpha Property, CountryCode Property, RegionName Property, Subdivision Property

IPAddress Property

Gets the external IP address for the local system.

Syntax

object.**IPAddress**

Remarks

This property returns a string which specifies the external IP address for the local system. If the system has been assigned multiple IP addresses, it reflects the address of the interface that was used to establish a connection with the SocketTools server. If the connection is made through a Virtual Private Network (VPN) it will use that assigned IP address. If a connection is made through a proxy server, the IP address may be address of the proxy rather than the local host, depending on how the connection is made.

This property cannot be assigned a value to query the location of different IP addresses. If the external IP address of the local system cannot be determined, this property will return an empty string.

Data Type

String

See Also

ASNumber Property, LocationId Property, Organization Property

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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

Latitude Property

Gets the latitude for the current location.

Syntax

object.**Latitude**

Remarks

This property returns a value which specifies the latitude of the location in decimal format. A positive value indicates a location that is north of the equator, while a negative value is a location that is south of the equator.

Data Type

Double

See Also

Coordinates Property, Longitude Property

LastUpdate Property

Gets the number of seconds since the last location update.

Syntax

object.**LastUpdate**

Remarks

This property returns an integer value which specifies the number of seconds since the last location query was performed by the control. A return value of zero indicates that the current location has not been updated.

Data Type

Integer (Int32)

See Also

Reset Method, Update Method

LocalTime Property

Gets the local time adjusted for the location's time zone.

Syntax

object.**LocalTime**

Remarks

This property returns a string which contains the current date and time at the location, adjusted for its time zone and whether or not it's in daylight savings time. The format of the date and time is determined by the current locale and system configuration.

Data Type

String

See Also

Timezone Property, TzOffset Property, TzShortName Property

LocationId Property

Gets a unique identifier for the current location.

Syntax

object.**LocationId**

Remarks

This property returns a string of hexadecimal characters which uniquely identifies the location for this computer system. This value is used internally by the location service, and may also be used by the application for its own purposes. If this value changes in subsequent queries, it indicates the external IP address for the local system has changed.

Data Type

String

See Also

ASNumber Property, IPAddress Property, Organization Property

Longitude Property

Gets the longitude for the current location.

Syntax

object.**Longitude**

Remarks

This property returns a value which specifies the longitude of the location in decimal format. A positive value indicates a location that is east of the prime meridian, while a negative value is a location that is west of the prime meridian.

Data Type

Double

See Also

Coordinates Property, Latitude Property

Organization Property

Gets the name of the organization that owns the network.

Syntax

object.**Organization**

Remarks

This property returns a string which identifies the organization associated with the local system's external IP address. For residential end-users this is typically the name of their Internet Service provider, however it may also identify a private company such as Microsoft, Google or Amazon. Because of the nature of how this information is updated, the organization names can change over time due to acquisitions or changes of ownership. If the owner of the network cannot be determined, this property may return an empty string.

Data Type

String

See Also

ASNumber Property, IPAddress Property, LocationId Property

PostalCode Property

Gets the postal code for the current location.

Syntax

object.**PostalCode**

Remarks

This property returns a string which contains the postal code associated with the area where the IP address is located. In the United States, this is a 5-digit numeric code. In Canada, this will contain the forward sortation area (the first three characters of the six character postal code). Local delivery portions of a postal code (such as the ZIP+4 code in the United States, or the local delivery unit in Canada) are not included.

This information will be most accurate within the geographic region of North America. Postal codes are locale specific, and this property may return an empty string if the postal code for the location cannot be determined.

Data Type

String

See Also

CityName Property, CountryName Property, RegionName Property, Subdivision Property

RegionCode Property

Gets the numeric region code for the current location.

Syntax

object.**RegionCode**

Remarks

This property returns an integer value which identifies the geographical region in which the external IP address is located. This value corresponds to the name returned by the **RegionName** property. The numeric values use the UN M49 standard established by the United Nations Statistics Division.

Data Type

Integer (Int32)

See Also

CityName Property, CountryName Property, RegionName Property, Subdivision Property

RegionName Property

Gets the region name for the current location.

Syntax

object.**RegionName**

Remarks

This property returns a string which identifies the region in which the external IP address is located. This refers to a broad geographical area, such as "North America" or "Southeast Asia" and uses the conventions for supranational regions as defined by UN M49 codes. These names will always be in English, regardless of the current system locale.

Data Type

String

See Also

CityName Property, CountryName Property, RegionCode Property, Subdivision Property

Subdivision Property

Gets the subdivision name for the current location.

Syntax

object.**Subdivision**

Remarks

This property returns a string which identifies the geopolitical subdivision within a country where the external IP address is located. In the United States, this will contain the full name of the state or commonwealth. In Canada, this will contain the name of the province or territory. These names will always be in English, regardless of the current system locale.

If a subdivision name does not exist for the location, this property will return an empty string.

The **SubdivisionCode** property will return a standardized abbreviation for the area. For example, in the United States it would return the two-character code for the state.

Data Type

String

See Also

CityName Property, CountryName Property, RegionName Property, SubdivisionCode Property

SubdivisionCode Property

Gets the alphanumeric code for the current subdivision.

Syntax

object.**SubdivisionCode**

Remarks

This property returns a string which is either a two- or three-letter code which identifies a geopolitical subdivision within the country where the external IP address is located. These codes are defined by the ISO 3166-2 standard. For example, the code for the state of California in the United States is "CA".

For international specificity, these subdivision codes are often combined with the value of the ISO alpha-2 code returned by the **CountryAlpha** property. For example, to identify the state of California, you could combine with the alpha-2 code for the United States, creating "US-CA" as an identifier.

If a subdivision code does not exist for the location, this property will return an empty string.

Data Type

String

See Also

CountryAlpha Property, CountryCode Property, RegionCode Property, Subdivision Property

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the number of seconds to wait until a location query fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting this property specifies the number of seconds until a request for location information fails and returns an error. This includes the amount of time spent attempting to connect to the location service. If a connection cannot be established within the given time period, the operation will fail.

Data Type

Integer (Int32)

See Also

Update Method

Timezone Property

Gets the time zone for the current location.

Syntax

object.**Timezone**

Remarks

This property returns a string which specifies the full time zone name in which the external IP address is located. These names are defined by the Internet Assigned Numbers Authority (IANA) and have values like "America/Los_Angeles" and "Europe/London". These time zones may also be defined as "Etc/GMT+10" if there is not a regional name associated with the time zone.

The **TzShortName** property will return an abbreviated time zone name, such as "PST".

Data Type

String

See Also

LocalTime Property, TzOffset Property, TzShortName Property

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
webTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
webTraceError	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.
webTraceWarning	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.
webTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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

TzOffset Property

Gets the time zone offset in seconds for the current location.

Syntax

object.**TzOffset**

Remarks

This property returns an integer which specifies the number of seconds east or west of the prime meridian (UTC). A positive value indicates a time zone that is east of the prime meridian and a negative value indicates a time zone that is west of the prime meridian. For example, the Pacific time zone in the western United States during daylight savings time would have a value of -25200 (7 hours).

Data Type

Integer (Int32)

See Also

LocalTime Property, Timezone Property, TzShortName Property

TzShortName Property

Gets an abbreviated time zone name for the current location.

Syntax

object.**TzShortName**

Remarks

This property returns a string which specifies the abbreviated time zone code in which the external IP address is located. If daylight savings time is used within the time zone, then this value can change based on whether or not daylight savings is in effect. For example, if the IP address is located within the Pacific time zone in the United States, this will return "PDT" when daylight savings is in effect and "PST" when it is not.

If the time zone code cannot be determined for this location, a value such as "UTC+9" may be returned, indicating the number of hours ahead or behind UTC.

Data Type

String

See Also

LocalTime Property, Timezone Property, TzOffset Property

Version Property

Return the current version of the object.

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

Web Location Control Methods

Method	Description
Initialize	Initialize the control and validate the runtime license key
Reset	Reset the internal state of the control
Uninitialize	Uninitialize the control and release any system resources that were allocated
Update	Update the current location information for the local system

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 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 ipLocation = CreateObject("SocketTools.WebLocation.11")

nError = ipLocation.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize 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, open network connections will be closed and any resources allocated by the control will be released.

See Also

Initialize Method, Uninitialize 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 resets the internal state of the control and releases system resources allocated for this class instance. 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

Update Method

Update the current location information for the local system.

Syntax

object.**Update**

Parameters

None.

Return Value

A value of True is returned if the current location was updated successfully. A value of False indicates that the current location could not be determined. The **LastError** property can be used to identify the specific cause of the failure.

Remarks

This method causes the control to query the location service to obtain current information about the physical location of the computer system based on its external IP address. The location data is cached and additional queries are only performed if it detects the external IP address for the local system has changed.

See Also

LastUpdate Property, Timeout Property

Web Location Control Events

Event	Description
OnError	This event is generated when a control error occurs

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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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

Web Storage Control

Application storage and data management services.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	WebStorageCtl.WebStorage
File Name	CSWEBX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.WebStorage.11
ClassID	28DA32F6-1F2F-43C8-8C13-FDD28245F771
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None

Overview

The Web Storage control enables an application to securely store and manage data remotely.

This control uses SocketTools Web Services and will only function if there is an active Internet connection and the local system is capable of establishing a secure connection to our servers.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

Web Storage Control Properties

Property	Description
AccountId	Gets a value that specifies the storage account ID
AppId	Gets the application identifier associated with the storage container
IsBlocked	Determine if a blocking storage operation is in progress
IsConnected	Determine if the application is connected to the storage server
IsInitialized	Determine if the control has been initialized
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Returns a description of the last error to occur
ObjectAttributes	Gets the attributes for the current object
ObjectContent	Gets the content type for the current object
ObjectCreated	Gets the date and time the current object was created
ObjectDigest	Gets the SHA-256 digest value for the current object
ObjectId	Gets the unique identifier for the current object
ObjectLabel	Gets the label for the current object
ObjectLimit	Gets the maximum number of objects that may be created
ObjectModified	Gets the date and time the current object was last modified
ObjectSize	Gets the size of the current object in bytes
StorageFree	Gets the total number of bytes available to store new objects
StorageId	Gets a unique identifier for the current storage container
StorageLimit	Gets the maximum number of bytes of storage available
StorageObjects	Gets the current number of objects stored for your account
StorageType	Gets the current storage container type identifier
StorageUsed	Gets the current number of bytes of storage used for your account
ThrowError	Enable or disable error handling by the container of the control
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
TransferBytes	Gets the number of bytes transferred from the storage server
TransferRate	Gets the current data transfer rate in bytes per second
TransferTime	Gets the number of seconds elapsed for the current data transfer
Version	Return the current version of the object

AccountId Property

Gets a value that specifies the storage account ID associated with the development license.

Syntax

object.**AccountId**

Remarks

This property is a string that uniquely identifies the web services account that is associated with the session. The account ID corresponds with your product serial number and runtime license key, but it is not identical to either of those values.

If you are using an evaluation license, the account ID is temporary and only valid during the evaluation period. After the evaluation period has expired, the account ID is revoked and objects stored using this ID will be deleted. It is not recommended that you store critical application data using an evaluation license.

Data Type

String

See Also

AppId Property, ObjectId Property, StorageId Property

AppId Property

Gets the application identifier associated with the storage container.

Syntax

object.**AppId**

Remarks

This property returns the current application ID. The application ID is a string that uniquely identifies the application and can only contain letters, numbers, the period and the underscore character. The default value for this property is **`SocketTools.Storage.Default`**.

You can register a unique identifier for your application using the **RegisterId** method.

Data Type

String

See Also

ObjectId Property, RegisterId Method, ValidateId Method, UnregisterId Method

IsBlocked Property

Determine if a blocking storage operation is in progress.

Syntax

object.**IsBlocked**

Remarks

The **IsBlocked** property is used to determine if a storage operation is currently in progress. For example, this property will return True if checked within an **OnProgress** event handler when the **GetFile** or **PutFile** method has been called.

Data Type

Boolean

See Also

GetData Method, GetFile Method, PutData Method, PutFile Method, OnProgress Event

IsConnected Property

Determine if the application is connected to the storage server.

Syntax

object.**IsConnected**

Remarks

The **IsConnected** property is used to determine if the application is connected to the storage server. This property will return **False** if the **Open** method has not been called to open a storage container, or if the connection to the server has been terminated. A value of **True** indicates that the storage container has been opened and there is a valid connection to the server.

The client does not maintain a continuous, persistent connection with the storage server. The connection may be closed and reopened internally as needed. If the client session has been idle for a period of time, this property can return **False**. If the **LastError** property returns **stErrorNotConnected** it means the client session is valid, however it not currently connected to the storage server. The next call to store or retrieve an object will the cause the client to reconnect automatically.

Data Type

Boolean

See Also

LastError Property, Close Method, Open 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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

ObjectAttributes Property

Gets the attributes for the current object.

Syntax

object.**ObjectAttributes**

Remarks

This property returns an integer value that specifies the attributes for the current storage object. The object attributes are comprised of one or more bit flags.

Value	Description
webObjectDefault	Default object attributes. This value is used to indicate the object can be modified, or that the attributes for a previously existing object should not be changed.
webObjectNormal	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
webObjectReadOnly	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
webObjectHidden	A hidden object. Objects with this attribute are not returned when enumerated using the **FindFirst** and **FindNext** methods. The object can only be accessed directly when specifying its label.

Data Type

Integer (Int32)

See Also

Exists Method, FindFirst Method, FindNext Method

ObjectContent Property

Gets the content type for the current object.

Syntax

object.**ObjectContent**

Remarks

This property returns a string which specifies the MIME content type for the current storage object. The content type is typically determined by the object label and evaluating the contents of the object. It is also possible for the application to explicitly specify the content type of the object when it is created.

The object content type will always be in the format **type/subtype** where the type specifies a common media type (e.g.: text, audio, video, etc.) and subtype specifies the specific content. The most common content type for text files is **text/plain**. If the content type is unknown, the default content type is **application/octet-stream**.

Text objects may also optionally include the character encoding as part of the content type. For example, if an object contains UTF-8 encoded text, the content type may be returned as **text/plain; charset=utf-8**. If your application is parsing the content types, you must check if a character encoding was also included in the value. Text objects that do not specify an encoding either contain ASCII or text which uses the system code page. Unicode text will always be stored using UTF-8 encoding.

Data Type

String

See Also

Exists Method, PutData Method, PutFile Method

ObjectCreated Property

Gets the date and time the current object was created.

Syntax

object.**ObjectCreated**

Remarks

This property returns a string which specifies the date and time the current object was created. The format of the date and time string is determined by the system configuration and current locale.

Data Type

String

See Also

ObjectModified Property, Exists Method, GetData Method, GetFile Method

ObjectDigest Property

Gets the SHA-256 digest value for the current object.

Syntax

object.**ObjectDigest**

Remarks

This property returns a string value which specifies the digest of the object contents, computed using an SHA-256 hash. The digest value is always represented as a string of hexadecimal numbers that is exactly 64 characters long. It is important to note that even a zero-length object will have a digest, which is the standard SHA-256 NULL hash value.

Data Type

String

See Also

ObjectId Property, ObjectContent Property, ObjectSize Property, Exists Method

ObjectId Property

Gets the unique identifier for the current object.

Syntax

object.**ObjectId**

Remarks

This property returns a unique identifier associated with the current object. Object IDs are guaranteed to be unique for each storage object that is created by the application.

Data Type

String

See Also

ObjectContent Property, ObjectDigest Property, ObjectLabel Property, Exists Method

ObjectLabel Property

Gets the label for the current object.

Syntax

object.**ObjectLabel**

Remarks

This property returns a string which specifies the label assigned to the current object by the application. Object labels are case-sensitive and must be unique for each object. An application uses labels to reference an object with a human-recognizable name, rather than referencing them by their object ID.

Object labels are similar to Windows file names, except they are case-sensitive. The maximum length of a label string is 511 characters. Leading and trailing whitespace (spaces, tabs, linebreaks, etc.) are ignored in label names.

Illegal characters include ASCII and Unicode control characters 0 through 31, single quotes (39), double quotes (34), less than symbol (60), greater than symbol (62), pipe (124), asterisk (42) and question mark (63). It is not possible to embed null characters in the label name.

Label names may contain forward slash (47) characters and backslash (92) characters, however it is important to note that objects are not stored in a hierarchical structure. An application can create its own folder-like structure to the labels it creates, but this structure is not imposed or enforced by the control.

Labels can contain Unicode characters which are internally encoded as UTF-8.

Data Type

String

See Also

ObjectContent Property, ObjectDigest Property, ObjectId Property, Exists Method, ValidateLabel Method

ObjectLimit Property

Gets the maximum number of objects that may be created.

Syntax

object.**ObjectLimit**

Remarks

This property returns an integer value which specifies the maximum number of storage objects that may be created. In addition to the limit on the total amount of storage that may be used, there is a limit on the total number of objects that may be created by all applications.

Data Type

Integer (Int32)

See Also

StorageFree Property, StorageLimit Property, StorageObjects Property StorageUsed Property

ObjectModified Property

Gets the date and time the current object was last modified.

Syntax

object.**ObjectModified**

Remarks

This property returns a string which specifies the date and time the current object was last modified. The format of the date and time string is determined by the system configuration and current locale.

Data Type

String

See Also

ObjectCreated Property, Exists Method, GetData Method, GetFile Method

ObjectSize Property

Gets the size of the current object in bytes.

Syntax

object.**ObjectSize**

Remarks

This property returns a value that specifies the size of the current storage object in bytes.

Data Type

Integer (Int32)

See Also

ObjectLimit Property StorageFree Property, StorageLimit Property, StorageUsed Property

StorageFree Property

Gets the total number of bytes available to store new objects.

Syntax

object.**StorageFree**

Remarks

This property returns a value which specifies the number of bytes available for the storage of new objects. This value reflects the total amount of available storage across all applications registered with the development account. If this value is zero, your storage account has reached its storage limit.

If your storage quota has been exceeded, either because the total number of objects or the total bytes of storage has reached their limit, your applications will be unable to create new objects. Your application can continue to access existing objects, regardless of your current quota limits.

To free storage space, use the **Delete** method to delete individual storage objects that are no longer needed by your application, or use the **DeleteAll** method to delete all objects in the current container.

Data Type

Double

See Also

ObjectLimit Property, StorageLimit Property, StorageObjects Property, StorageUsed Property

StorageId Property

Gets a unique identifier for the current storage container.

Syntax

object.**StorageId**

Remarks

This property returns a string which identifies the current storage container opened with the **Open** method. The storage ID is associated with your development license and is guaranteed to be a unique value. If no storage container has been opened, this property will return a zero length string.

Data Type

String

See Also

AppId Property, ObjectId Property, Open Method

StorageLimit Property

Gets the maximum number of bytes of storage available.

Syntax

object.**StorageLimit**

Remarks

This property returns a value which specifies the maximum number of bytes of data storage available. This limit applies to all applications registered with the development account. In addition to limits on the total number of bytes that can be stored, there are also limits on the total number objects which may be created, and the individual size of each object.

Storage quota limits are assigned for each SocketTools development account. Accounts that are created with an evaluation license have much lower quota limits than a standard account and should be used for testing purposes only. After the evaluation period has ended, all objects stored using the evaluation license will be deleted.

This value does not represent limits on the storage used by a specific application. Quotas limits apply to all applications that are registered with the development account, which is identified with the runtime license key passed to the **Initialize** method.

If your storage quota has been exceeded, either because the total number of objects or the total bytes of storage has reached their limit, your applications will be unable to create new objects. Your application can continue to access existing objects, regardless of your current quota limits.

To free storage space, use the **Delete** method to delete individual storage objects that are no longer needed by your application, or use the **DeleteAll** method to delete all objects in the current container.

Data Type

Double

See Also

ObjectLimit Property, StorageLimit Property, StorageObjects Property, StorageUsed Property

StorageObjects Property

Gets the current number of objects stored for your account.

Syntax

object.**StorageObjects**

Remarks

This property returns an integer value which specifies the number of storage objects allocated for the account. This value may not exceed the total number of objects specified by the **ObjectLimit** property.

Data Type

Integer (Int32)

See Also

ObjectLimit Property, StorageLimit Property, StorageUsed Property

StorageType Property

Gets the current storage container type identifier.

Syntax

object.**StorageType**

Remarks

This property returns an integer value which identifies the current storage container type. It may be one of the following values. If no storage container is currently open, this property will return a value of zero.

Value	Description
webStorageGlobal	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
webStorageDomain	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
webStorageMachine	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
webStorageUser	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

The storage type specifies the type of container that objects will be stored in. You can think of the storage containers as special folders which store individual objects. In most cases, we recommend using **webStorageGlobal** which means that stored objects will be accessible to all users of your application. However, you can limit access to the stored objects based on the local domain, local machine ID or the current user SID.

If you specify anything other than global storage, objects can be orphaned if the system configuration changes. For example, if **webStorageMachine** is specified, the objects that are stored there can only be accessed from that computer system. If the system is reconfigured (for example, the boot volume formatted and Windows is reinstalled) the unique identifier for that system will change and the previous objects that were stored by your application can no longer be accessed.

It is advisable is to store critical application data and configuration information using

webStorageGlobal and use other non-global storage containers for configuration information that is unique to that system and/or user which is not critical and can be easily recreated.

Data Type

Integer

See Also

StorageId Property, Open Method

StorageUsed Property

Gets the current number of bytes of storage used for your account.

Syntax

object.**StorageUsed**

Remarks

This property returns a value which specifies the total number of bytes of data allocated for all storage objects. This value may not exceed the total number of bytes of storage available, which is returned by the **StorageLimit** property.

This value does not represent the storage used by a specific application. This property returns the amount of storage used by all applications that are registered with the development account, which is identified with the runtime license key passed to the **Initialize** method.

Data Type

String

See Also

ObjectLimit Property, StorageFree Property, StorageLimit Property, StorageObjects Property

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error.

Note that the **Timeout** property also determines the amount of time the control will spend attempting to connect to a server. If a connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TransferBytes Property

Gets the number of bytes transferred from the storage server.

Syntax

object.**TransferBytes**

Remarks

The **TransferBytes** property returns the number of bytes that have been copied to or from the storage server. If this property is read while a transfer is ongoing, the property returns the number of bytes that have been copied up to that point. If read after a transfer has completed, the total number of bytes copied is returned.

This property value is reset with every data transfer.

Data Type

Integer (Int32)

See Also

TransferRate Property, TransferTime Property, OnProgress Event

TransferRate Property

Gets the current data transfer rate in bytes per second.

Syntax

object.**TransferRate**

Remarks

The **TransferRate** property returns the rate at which the file data is being transferred, expressed in bytes per second. If this property is read while a transfer is ongoing, it returns the current average transfer rate.

If this property is read after the transfer has completed, it returns the final transfer rate which is calculated as the total number of bytes transferred divided by the number of seconds to complete the transfer. This property value is reset with every data transfer.

Data Type

Integer (Int32)

See Also

TransferBytes Property, TransferTime Property, OnProgress Event

TransferTime Property

Gets the number of seconds elapsed for the current data transfer.

Syntax

object.**TransferTime**

Remarks

The **TransferTime** property returns the number of seconds that have elapsed since the data transfer started. If the property is read after the transfer has completed, it returns the total number of seconds it took to transfer the object data.

This property value is reset with every data transfer.

Data Type

Integer (Int32)

See Also

TransferBytes Property, TransferRate Property, OnProgress Event

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
webTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
webTraceError	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.
webTraceWarning	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.
webTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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 object.

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

Web Storage Control Methods

Method	Description
Cancel	Cancel the current storage operation
Close	Close the open storage container
CompareData	Compare the data in a buffer with a stored object
CompareFile	Compare the data in a file with a stored object
Copy	Copy the contents of a stored object to another container
Delete	Delete a stored object from the container
DeleteAll	Delete all stored objects from the current container
Exists	Determine if a specific stored object exists in the container
FindFirst	Find the first stored object that matches a label or content type
FindNext	Find the next stored object that matches a label or content type
GetData	Download the data in a stored object to a string or byte array buffer
GetFile	Download a stored object to a file on the local system
Initialize	Initialize the control and validate the runtime license key
Move	Move a stored object to another container
Open	Open a storage container
PutData	Upload the data in a string or byte array buffer to the storage container
PutFile	Upload a local file to the storage container
RegisterId	Register a new application identifier with the storage service
Rename	Change the label of an existing storage object
Reset	Reset the internal state of the control
Uninitialize	Uninitialize the control and release any system resources that were allocated
UnregisterId	Unregister a previously registered application identifier
ValidateId	Check an application identifier to ensure it is valid and exists
ValidateLabel	Validate an object label to ensure it uses allowed characters

Cancel Method

Cancels the current storage operation.

Syntax

object.**Cancel**

Parameters

None.

Return Value

None.

Remarks

The **Cancel** method cancels the current storage operation in the current thread. This is typically used inside an **OnProgress** event handler, causing the blocking method to return to the caller with an error indicating that the current operation was canceled. If a storage operation is not currently in progress, calling this method will have no effect.

See Also

GetData Method, GetFile Method, PutData Method, PutFile Method, Reset Method, OnProgress Event

Close Method

Close the storage container.

Syntax

object.**Close**

Parameters

None.

Return Value

A value of True is returned if the storage container was opened. Otherwise, a value of False is returned and the **LastError** property will return the specific cause of the failure.

Remarks

The **Close** method should be called after all operations using the storage container have completed. The access token granted to the application will be released and the memory allocated for the session cache will be freed.

See Also

Open Method, Reset Method

CompareData Method

Compare the data in a buffer with a stored object.

Syntax

object.**CompareData**(*ObjectLabel*, *Buffer*)

Parameters

ObjectLabel

A string which specifies the label of the object that should be compared against the contents of the buffer.

Buffer

A string or byte array which contains the data that should be compared against the contents of the object. If the buffer contains binary data (particularly data that contains embedded null bytes) this parameter must be a byte array. String values should only be used for text comparisons.

Return Value

A value of True is returned if the contents of the buffer matches the contents of the specified object. Otherwise, a value of False is returned and the **LastError** property will return the specific cause of the failure.

Remarks

The **CompareData** method performs a binary comparison of the data in the specified buffer with the contents of the storage object on the server. The amount of data in the buffer must match the size of the stored object exactly, or this method will fail. Partial comparisons are not supported by this method.

If you need to compare the contents of a file with a stored object, use the **CompareFile** method.

See Also

CompareFile Method, GetData Method, PutData Method

CompareFile Method

Compare the data in a file with a stored object.

Syntax

object.**CompareFile**(*ObjectLabel*, *LocalFile*)

Parameters

ObjectLabel

A string which specifies the label of the object that should be compared against the contents of the buffer.

LocalFile

A string which specifies the name of the file to compare against the contents of the stored object. If no path is specified in the file name, the current working directory will be used.

Return Value

A value of True is returned if the contents of the file matches the contents of the specified object. Otherwise, a value of False is returned and the **LastError** property will return the specific cause of the failure.

Remarks

The **CompareFile** method performs a binary comparison of the data in the file with the contents of the storage object on the server. The contents of the file must be identical to the contents of the stored object or the method will fail.

If you need to compare the contents of a file with a string or byte array, use the **CompareData** method.

See Also

CompareData Method, GetData Method, PutData Method

Copy Method

Copy the contents of a stored object to another container.

Syntax

object.**Copy**(*OldLabel*, *NewLabel* [, *StorageType*])

Parameters

OldLabel

A string which specifies the label of the object that should be copied.

NewLabel

A string which specifies the new label for the copied object.

StorageType

A numeric value that identifies the storage container type. One of the following values should be specified. If this parameter is omitted, the object will be copied within the current container.

Value	Description
webStorageGlobal	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
webStorageDomain	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
webStorageMachine	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
webStorageUser	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

Return Value

A value of True is returned if the object was copied. Otherwise, a value of False is returned and the **LastError** property will return the specific cause of the failure.

Remarks

The **Copy** method is used to create a copy of an existing storage object. It may be used to duplicate

an object with a different label, or it may be used to copy the object to a new storage container type. For example, it can copy an object originally created in the **webStorageUser** container to a new object stored in the **webStorageMachine** container.

Copied objects are assigned their own unique ID and are not linked to one another. Any subsequent changes made to the original object will not affect the copied object. Attempting to copy an object to itself or another existing object will result in an error.

This method updates the current object. Various properties such as **ObjectId** and **ObjectLabel** will reflect the values associated with the new, copied object and not the original object it was copied from.

See Also

Delete Method, Move Method, Rename Method

Delete Method

Delete a stored object from the container.

Syntax

object.**Delete**(*ObjectLabel*)

Parameters

ObjectLabel

A string which specifies the label of the object that should be deleted.

Return Value

A value of True is returned if the object was deleted. Otherwise, a value of False is returned and the **LastError** property will return the specific cause of the failure.

Remarks

The **Delete** method is used to delete a stored object from the container. This method permanently deletes the storage object and its associated data from the server. Deleted objects cannot be recovered by the application. To remove all objects stored in the container, use the **DeleteAll** method.

See Also

Copy Method, DeleteAll Method, Move Method, Rename Method

DeleteAll Method

Delete all stored objects in the container.

Syntax

object.**DeleteAll**

Parameters

None

Return Value

A value of True is returned if all objects were deleted from the current container. Otherwise, a value of False is returned and the **LastError** property will return the specific cause of the failure.

Remarks

The storage container contains information for each of the objects that have been stored by the application. Each of these objects are associated with the application ID and the storage type that was specified when calling the **Open** method. The **DeleteAll** method instructs the server to remove all objects in the container, resetting it back to its initial state.

Exercise caution when using this method. The operation is immediate and the objects that are stored in the container are permanently deleted. They cannot be recovered by your application.

See Also

Copy Method, Delete Method, Move Method, Open Method, Rename Method

Exists Method

Determine if a specific stored object exists in the container.

Syntax

object.**Exists**(*ObjectLabel*)

Parameters

ObjectLabel

A string which specifies the label of the object.

Return Value

A value of True is returned if the object exists. Otherwise, a value of False is returned and the **LastError** property will return the specific cause of the failure.

Remarks

The **Exists** method is used to check for the existence of a stored object in the current container. If the object exists, various properties that return information about the current object, such as **ObjectId** and **ObjectSize** will be updated to return the metadata associated with the object.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting information about an object, your application must specify the label name exactly as it was created. The object label cannot contain wildcard characters.

To obtain information about how much storage your applications are using and the total number of stored objects, use the **StorageUsed** and **StorageObjects** properties.

If you wish to enumerate all of the stored objects within the container, use the **FindFirst** and **FindNext** methods.

See Also

ObjectId Property, ObjectModified Property, ObjectSize Property, StorageObjects Property, StorageUsed Property, FindFirst Method, FindNext Method

FindFirst Method

Find the first stored object that matches a label or content type.

Syntax

object.**FindFirst**([*MatchLabel*], [*ContentType*])

Parameters

MatchLabel

A string which specifies the value to match against the object labels in the container. The string may contain wildcard characters similar to those use with the Windows filesystem. A "?" character matches any single character, and "*" matches any number of characters in the label. If this parameter is omitted or an empty string, all objects in the container will be matched.

ContentType

A string which specifies the content type of the objects to be enumerated. If this parameter is omitted or an empty string, the content type is ignored and all matching objects are returned. If a content type is specified, it must be a valid MIME media content type designated using the **type/subtype** nomenclature.

Return Value

A value of True is returned if a matching object exists. Otherwise, a value of False is returned and the **LastError** property will return the specific cause of the failure.

Remarks

The **FindFirst** method returns information about the first object that matches a given label, content type or both. It is used in conjunction with the **FindNext** method to enumerate all of the matching objects in the storage container.

If a matching object exists, various properties that return information about the current object, such as **ObjectId** and **ObjectSize** will be updated to return the metadata associated with the object.

See Also

Exists Method, FindNext Method

FindNext Method

Find the next stored object that matches a label or content type.

Syntax

object.**FindNext**

Parameters

None.

Return Value

A value of True is returned if a matching object exists. If there are no more matching objects, or an error occurs, a value of False is returned and the **LastError** property will return the specific cause of the failure. If the **LastError** property returns a value of zero, then no error occurred and there are no additional matching objects.

Remarks

The **FindNext** method returns information about the next object that matches a given label, content type or both. This method may only be called after the **FindFirst** method is called, otherwise it will fail.

If a matching object exists, various properties that return information about the current object, such as **ObjectId** and **ObjectSize** will be updated to return the metadata associated with the object.

See Also

Exists Method, FindFirst Method

GetData Method

Download the data in a stored object to a string or byte array buffer.

Syntax

object.**GetData**(*ObjectLabel*, *Buffer*, [*Length*])

Parameters

ObjectLabel

A string which specifies the label of the object that should be retrieved from the server.

Buffer

This parameter specifies the buffer that will contain the object data when the method returns. If the variable is a String type, then the data will be returned as a string of characters. This is the most appropriate data type to use if the object only contains text. If the object contains binary data, it is recommended that a Byte array variable be specified as the argument to this method.

Length

An optional integer argument that will contain the number of bytes copied into the buffer when the method returns.

Return Value

A value of True is returned on success. If an error occurs, a value of False is returned and the **LastError** property will return the specific cause of the failure..

Remarks

The **GetData** method transfers data from a stored object to the specified buffer. This method will cause the current thread to block until the data transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

When this method returns, various properties such as **ObjectId** and **ObjectLabel** will be updated to reflect the values associated with the object that was downloaded from the server.

See Also

GetFile Method, PutData Method, PutFile Method, OnProgress Event

GetFile Method

Download the data in a stored object to a local file.

Syntax

object.**GetFile**(*LocalFile*, *ObjectLabel*)

Parameters

LocalFile

A string which specifies the name of the local file that will be created or overwritten with the contents of the storage object. If a path is not specified, the file will be created in the current working directory.

ObjectLabel

A string which specifies specifies the label of the object that should be retrieved from the server.

Return Value

A value of True is returned on success. If an error occurs, a value of False is returned and the **LastError** property will return the specific cause of the failure..

Remarks

The **GetFile** method downloads data from a stored object to a local file. This method will cause the current thread to block until the data transfer completes, a timeout occurs or the transfer is canceled. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

When this method returns, various properties such as **ObjectId** and **ObjectLabel** will be updated to reflect the values associated with the object that was downloaded from the server.

See Also

GetData Method, PutData Method, PutFile Method, OnProgress Event

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 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 ipLocation = CreateObject("SocketTools.WebLocation.11")

nError = ipLocation.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize Method

Move Method

Move a stored object to another container.

Syntax

object.**Move**(*OldLabel*, *NewLabel* [, *StorageType*])

Parameters

OldLabel

A string which specifies the label of the object that should be moved.

NewLabel

A string which specifies the new label for the object being moved.

StorageType

A numeric value that identifies the storage container type. One of the following values should be specified. If this parameter is omitted, the object will be moved within the current container.

Value	Description
webStorageGlobal	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
webStorageDomain	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
webStorageMachine	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
webStorageUser	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

Return Value

A value of True is returned if the object was moved. Otherwise, a value of False is returned and the **LastError** property will return the specific cause of the failure.

Remarks

The **Move** method is used to move an existing storage object to a new container. For example, it can

move an object originally created in the **webStorageUser** container to the **webStorageMachine** container. Using this method to move an object within the same container is effectively the same as calling the **Rename** method.

This method updates the current object. Various properties such as **ObjectId** and **ObjectLabel** will reflect the values associated with the object which has been moved.

See Also

Copy Method, Delete Method, Rename Method

Open Method

Open a storage container.

Syntax

object.**Open**([*AppId*], [*StorageType*], [*Timeout*])

Parameters

AppId

A string which specifies the application ID for the storage container. The application ID is a string that uniquely identifies the application and can only contain letters, numbers, the period and the underscore character. If this parameter is omitted or an empty string, the default identifier `SocketTools.Storage.Default` will be used.

StorageType

A numeric value that identifies the storage container type. One of the following values should be specified. If this parameter is omitted, global storage will be used by default.

Value	Description
webStorageGlobal	Global storage. Objects stored using this storage type are available to all users. Any changes made to objects using this storage type will affect all users of the application. Unless there is a specific need to limit access to the objects stored by the application to specific domains, local machines or users, it is recommended that you use this storage type when creating new objects.
webStorageDomain	Local domain storage. Objects stored using this storage type are only available to users in the same local domain, as defined by the domain name or workgroup name assigned to the local system. If the domain or workgroup name changes, objects previously stored using this storage type will not be available to the application.
webStorageMachine	Local machine storage. Objects stored using this storage type are only available to users on the same local machine. The local machine is identified by unique characteristics of the system, including the boot volume GUID. Objects previously stored using this storage type will not be available on that system if the boot disk is reformatted.
webStorageUser	Current user storage. Objects stored using this storage type are only available to the current user logged in on the local machine. The user identifier is based on the Windows user SID that is assigned when the user account is created. If the user account is deleted, the objects previously stored using this storage type will not be available to the application.

Timeout

The number of seconds that the client will wait for a response before failing the operation. If this argument is not specified, the value of the **Timeout** property will be used as the default.

Return Value

A value of True is returned if the storage container was opened. Otherwise, a value of False is returned

and the **LastError** property will return the specific cause of the failure.

Remarks

The **Open** method opens the specified storage container and requests an access token for the application. This method that must be called prior to accessing any stored objects.

The application ID is a string that uniquely identifies the application requesting the access and must have been previously registered with the server by calling the **RegisterId** method. If the *AppId* parameter is omitted or an empty string, the a default internal ID will be used which is allocated for each storage account. You can use this default ID if you wish to share data between all of the applications you create.

The storage type specifies the type of container that objects will be stored in. In most cases, we recommend using **webStorageGlobal** which means that stored objects will be accessible to all users of your application. However, you can limit access to the stored objects based on the local domain, local machine ID or the current user SID.

If you specify anything other than global storage, objects can be orphaned if the system configuration changes. For example, if **webStorageMachine** is specified, the objects that are stored there can only be accessed from that computer system. If the system is reconfigured (for example, the boot volume formatted and Windows is reinstalled) the unique identifier for that system will change and the previous objects that were stored by your application will no longer be accessible.

It is advisable is to store critical application data and configuration information in the **webStoragebGlobal** container and use other non-global storage containers for configuration information that is unique to that system and/or user which is not critical and can be easily recreated.

See Also

AppId Property, StorageType Property, Timeout Property, Close Method, RegisterId Method

PutData Method

Upload the data in a string or byte array buffer to the storage container.

Syntax

object.**PutData**(*ObjectLabel*, *Buffer*, [*Length*], [*ContentType*], [*Attributes*])

Parameters

ObjectLabel

A string which specifies the label of the object that should be uploaded to the server. If an object with the same label already exists, it will be replaced with the contents of the buffer.

Buffer

This parameter specifies the buffer that contains the object data to be stored. If the variable is a String type, then the data will be stored as text. If the buffer contains binary data, the buffer must be specified as a Byte array.

Length

An optional integer argument that specifies the number of bytes to be stored. If this parameter is omitted, the length is determined by the length of the string or size of the byte array buffer provided by the caller.

ContentType

An optional string argument that identifies the contents of the buffer being stored. If this parameter is omitted, or specifies a zero-length string, the method will attempt to automatically determine the content type based on the object label and the contents of the buffer.

Attributes

An optional integer argument which specifies the attributes associated with the storage object. This value can be a combination of one or more of the following bit flags using a bitwise OR operation. If this parameter is omitted, the default attribute **webObjectNormal** will be used.

Value	Description
webObjectNormal	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
webObjectReadOnly	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
webObjectHidden	A hidden object. Objects with this attribute are not returned when enumerated using the **FindFirst** and **FindNext** methods. The object can only be accessed directly when specifying its label.

Return Value

A value of True is returned on success. If an error occurs, a value of False is returned and the **LastError** property will return the specific cause of the failure..

Remarks

The **PutData** method uploads the contents of a buffer to the current storage container. If an object exists with the same label, it will be replaced. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting information about an object, your application must specify the label name exactly as it was created using this method. The object label cannot contain wildcard characters.

If the label identifies an object that already exists in the container, and that object was created with the **webObjectReadOnly** attribute, this method will fail. To replace a read-only object, the application must explicitly move, rename or delete the existing object.

If a content type is provided, it must specify a valid MIME media type and subtype. For example, normal text has a content type of **text/plain** while XML-formatted text would have a content type of **text/xml**. Data that contains unstructured binary data is typically identified as **application/octet-stream**. If you do not specify a content type, an appropriate content type will be determined automatically based on the label and the contents of the buffer.

If you wish to upload the contents of a file, use the **PutFile** method.

When this method returns, various properties such as **ObjectId** and **ObjectLabel** will be updated to reflect the values associated with the object that was created or replaced on the server.

See Also

GetData Method, GetFile Method, PutFile Method, OnProgress Event

PutFile Method

Upload a local file to the current storage container.

Syntax

object.**PutFile**(*LocalFile*, *ObjectLabel*, [*ContentType*], [*Attributes*])

Parameters

LocalFile

A string which specifies the name of the local file that will be uploaded to the current storage container. If a path is not specified, the file will be opened in the current working directory.

ObjectLabel

A string which specifies the label of the object that should be created. If an object with the same label already exists, it will be replaced with the contents of the file.

ContentType

An optional string argument that identifies the contents of the file being stored. If this parameter is omitted, or specifies a zero-length string, the method will attempt to automatically determine the content type based on the file name and contents.

Attributes

An optional integer argument which specifies the attributes associated with the storage object. This value can be a combination of one or more of the following bit flags using a bitwise OR operation. If this parameter is omitted, the default attribute **webObjectNormal** will be used.

Value	Description
webObjectNormal	A normal object that that can be read and modified by the application. This is the default attribute for new objects that are created by the application.
webObjectReadOnly	A read-only object that can only be read by the application. Attempts to modify or replace the contents of the object will fail. Read-only objects can be deleted.
webObjectHidden	A hidden object. Objects with this attribute are not returned when enumerated using the **FindFirst** and **FindNext** methods. The object can only be accessed directly when specifying its label.

Return Value

A value of True is returned on success. If an error occurs, a value of False is returned and the **LastError** property will return the specific cause of the failure..

Remarks

The **PutFile** method uploads the contents of a file to the current storage container. If an object exists with the same label, it will be replaced. During the transfer, the **OnProgress** event will fire periodically, enabling the application to update any user interface objects such as a progress bar.

Although storage object labels are similar to Windows file names, they are case-sensitive. When requesting information about an object, your application must specify the label name exactly as it was created using this method. The object label cannot contain wildcard characters.

If the label identifies an object that already exists in the container, and that object was created with the

webObjectReadOnly attribute, this method will fail. To replace a read-only object, the application must explicitly move, rename or delete the existing object.

If a content type is provided, it must specify a valid MIME media type and subtype. For example, normal text has a content type of **text/plain** while XML-formatted text would have a content type of **text/xml**. Data that contains unstructured binary data is typically identified as **application/octet-stream**. If you do not specify a content type, an appropriate content type will be determined automatically based on the file name and contents.

If you wish to upload the contents of a string or byte array, use the **PutData** method.

When this method returns, various properties such as **ObjectId** and **ObjectLabel** will be updated to reflect the values associated with the object that was created or replaced on the server.

See Also

GetData Method, GetFile Method, PutData Method, OnProgress Event

RegisterId Method

Register a new application identifier with the storage service.

Syntax

object.**RegisterId** (*AppId*)

Parameters

AppId

A string which identifies the application requesting access. If the application ID contains illegal characters, the method will fail. See the remarks below on the recommended method for identifying your application.

Return Value

A value of True is returned if the application ID was registered successfully. Otherwise, a value of False is returned and the **LastError** property will return the specific cause of the failure.

Remarks

The **RegisterId** method registers an application ID with the server which uniquely identifies the application that is requesting access to the storage container. The ID must only consist of ASCII letters, numbers, the period and underscore character. Whitespace characters and non-ASCII Unicode characters are not permitted. The maximum length of an application ID string is 63 characters.

It is recommended that you use a standard format for the application ID that consists of your company name, application name and optionally a version number. For example:

- MyCompany.MyApplication
- MyCompany.MyApplication.1

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

It is safe to call this method with an application ID that was previously registered. If the provided application ID has already been registered, this method will succeed.

If you no longer wish to use an application ID you have previously registered, you can call the **UnregisterId** method. Exercise caution when unregistering an application. This will cause all objects stored using that ID to be deleted by the storage server. Once an application ID has been unregistered, the operation is permanent. Calling **UnregisterId** and then **RegisterId** again using the same ID will force the system to create new access tokens for your application. You will not be able to regain access to the objects that were previously stored using that ID.

The application ID is intended to be an application defined human-readable string that uniquely identifies your application. If you want to obtain the internal storage ID associated with your application, get the value of the **StorageId** property. The storage ID is a fixed-length string of letters and numbers guaranteed to be unique across all applications that you register.

It is not required for your application to create a unique application ID. Each storage account has a default internal application ID named `SocketTools.Storage.Default`. This default ID is used if an

application-defined ID is not provided to the **Open** method. It is intended to identify storage available to all applications that you create.

See Also

AppId Property, Open Method, UnregisterId Method, ValidateId Method

Rename Method

Change the label of an existing storage object.

Syntax

object.**Rename**(*OldLabel*, *NewLabel*)

Parameters

OldLabel

A string which specifies the label of the object that should be moved.

NewLabel

A string which specifies the new label for the object being moved. An object with this label cannot already exist.

Return Value

A value of True is returned if the object was renamed. Otherwise, a value of False is returned and the **LastError** property will return the specific cause of the failure.

Remarks

The **Rename** method is used to change the label of an existing storage object within the current storage container. If you wish to move an object to a different container, use the **Move** method.

This method updates the current object. Various properties such as **ObjectId** and **ObjectLabel** will reflect the values associated with the object which has been renamed.

See Also

Copy Method, Delete Method, Move 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, open network connections will be closed and any handles allocated by the control will be released.

See Also

Initialize Method, Uninitialize 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. 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

UnregisterId Method

Unregister a previously registered application identifier.

Syntax

object.**UnregisterId** (*AppId*)

Parameters

AppId

A string which specifies the application ID to be deleted. If the application ID is a zero-length string or contains illegal characters, the method will fail.

Return Value

A value of True is returned if the application ID was deleted. Otherwise, a value of False is returned and the **LastError** property will return the specific cause of the failure.

Remarks

The **UnregisterId** method deletes the internal storage identifier associated with the application ID and revokes all access tokens that were granted for the application. This operation is immediate and permanent.

Exercise caution when using this method. This will permanently delete all objects that were stored for the specified application. Calling **UnregisterId** and then **RegisterId** again using the same ID will force the system to create new access tokens for your application. You will not be able to regain access to the objects that were previously stored using that ID.

This method cannot be used to unregister the default storage application identifier `SocketTools.Storage.Default`. If this ID is specified, the method will fail with an error indicating that the ID is invalid.

See Also

AppId Property, Open Method, RegisterId Method, ValidateId Method

ValidateId Method

Check an application identifier to ensure it is valid and exists.

Syntax

object.**ValidateId** (*AppId*)

Parameters

AppId

A string which specifies the application ID to be validated. If the application ID is a zero-length string or contains illegal characters, the method will fail.

Return Value

A value of True is returned if the application ID is valid. Otherwise, a value of False is returned and the **LastError** property will return the specific cause of the failure.

Remarks

The **ValidateId** method is used to determine if the specified application identifier is valid and has been previously registered using the **RegisterId** method. The ID must only consist of ASCII letters, numbers, the period and underscore character. Whitespace characters and non-ASCII Unicode characters are not permitted. The maximum length of an application ID string is 63 characters.

See Also

AppId Property, Open Method, RegisterId Method, UnregisterId Method

ValidateLabel Method

Validate an object label to ensure it uses allowed characters.

Syntax

object.**ValidateLabel** (*ObjectLabel*)

Parameters

ObjectLabel

A string which specifies the object label to be validated. This parameter cannot be a zero-length string.

Return Value

A value of True is returned if the object label is valid. Otherwise, a value of False is returned and the **LastError** property will return the specific cause of the failure.

Remarks

Object labels are similar to Windows file names, except they are case-sensitive. The maximum length of a label string is 511 characters. Leading and trailing whitespace (spaces, tabs, linebreaks, etc.) are ignored in label names.

Illegal characters include ASCII and Unicode control characters 0 through 31, single quotes (39), double quotes (34), less than symbol (60), greater than symbol (62), pipe (124), asterisk (42) and question mark (63). It is not possible to embed null characters in the label name.

Label names may contain forward slash (47) characters and backslash (92) characters, however it is important to note that objects are not stored in a hierarchical structure. An application can create its own folder-like structure to the labels it creates, but this structure is not imposed or enforced by the control.

Labels can contain Unicode characters which are internally encoded as UTF-8.

See Also

ObjectLabel Property, Exists Method

Web Storage Control Events

Event	Description
OnCancel	This event is generated when a storage operation is canceled
OnDownload	This event is generated when a stored object is downloaded
OnError	This event is generated when a control error occurs
OnProgress	This event is generated as a stored object is being transferred
OnTimeout	This event is generated when a storage operation times out
OnUpload	This event is generated when a stored object is uploaded

OnCancel Event

The **OnCancel** event is generated when a storage operation is canceled.

Syntax

Sub *object*_**OnCancel** ([*Index* **As Integer**])

Remarks

This event is generated when a storage operation, such as sending or receiving data, is canceled with the **Cancel** method.

See Also

Cancel Method, OnError Event, OnTimeout Event

OnDownload Event

The **OnDownload** event is generated when a storage object has been downloaded.

Syntax

Sub *object*_**OnDownload**([*Index* **As Integer**], **ByVal** *ObjectLabel* **As Variant**, **ByVal** *ObjectSize* **As Variant**)

Remarks

The **OnDownload** event is generated when a stored object has been successfully downloaded using either the **GetData** or **GetFile** methods. When this event occurs, the transfer has completed successfully and the downloaded object becomes the current object for the session. Prior to this event, the **OnProgress** event will occur periodically which provides information about the progress of the data transfer.

The ***ObjectLabel*** argument contains the label of object that has been downloaded, and the ***ObjectSize*** parameter contains the size of the downloaded object in bytes. Other values, such as the object digest (hash) and content type, can be determined by getting the values of the relevant properties.

See Also

GetData Method, GetFile Method, OnProgress 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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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 typically 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 during data transfer.

Syntax

Sub *object*_**OnProgress** ([*Index* **As Integer**], **ByVal** *ObjectLabel* **As Variant**, **ByVal** *BytesTotal* **As Variant**, **ByVal** *BytesCopied* **As Variant**, **ByVal** *Percent* **As Variant**)

Remarks

The **OnProgress** event is generated during the transfer of data between the application and storage server, indicating the amount of data exchanged. For transfers of 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:

ObjectLabel

A string value that specifies the name of the object that is being uploaded or downloaded.

BytesTotal

A numeric value that specifies the total amount of data being transferred in bytes. This value may be zero if the control cannot determine the total amount of data that will be copied. If the total number of bytes is less than 2 GiB, the value will be a **Long** (32-bit) integer. For very large transfers, it will be a **Double** floating-point value.

BytesCopied

A numeric value that specifies the number of bytes that have been transferred between the client and server. If the number of bytes copied is less than 2 GiB, the value will be a **Long** (32-bit) integer. For very large transfers, it will be a **Double** floating-point value.

Percent

The percentage of data that's been transferred, expressed as an integer value between 0 and 100, inclusive.

This event is only generated when data is transferred using the **GetData**, **GetFile**, **PutData** or **PutFile** methods.

See Also

TransferBytes Property, TransferRate Property, TransferTime Property, GetData Method, GetFile Method, PutData Method, PutFile Method

OnTimeout Event

The **OnTimeout** event is fired when a storage operation times out.

Syntax

Sub *object*_**OnTimeout** ([*Index* **As Integer**])

Remarks

The **OnTimeout** event is generated when a storage operation, such as uploading or downloading an object, times out.

See Also

Timeout Property, OnCancel Event, OnError Event

OnUpload Event

The **OnUpload** event is generated when a storage object has been uploaded.

Syntax

Sub *object*_**OnUpload**([*Index* **As Integer**], **ByVal** *ObjectLabel* **As Variant**, **ByVal** *ObjectSize* **As Variant**)

Remarks

The **OnUpload** event is generated when a stored object has been successfully uploaded using either the **PutData** or **PutFile** methods. When this event occurs, the transfer has completed successfully and the downloaded object becomes the current object for the session. Prior to this event, the **OnProgress** event will occur periodically which provides information about the progress of the data transfer.

The *ObjectLabel* argument contains the label of object that has been downloaded, and the *ObjectSize* parameter contains the size of the downloaded object in bytes. Other values, such as the object digest (hash) and content type, can be determined by getting the values of the relevant properties.

See Also

PutData Method, PutFile Method, OnProgress Event

Whois Protocol Control

Request registration information for an Internet domain name.

Reference

- Properties
- Methods
- Events
- Error Codes

Control Information

Object Name	WhoisClientCtl.WhoisClient
File Name	CSWHOX11.OCX
Version	11.0.2218.1855
ProgID	SocketTools.WhoisClient.11
ClassID	5FBB30C5-BEEB-45C0-965F-B989E543BC56
Threading Model	Apartment
Help File	CST11CTL.CHM
Dependencies	None
Standards	RFC 954

Overview

The Whois protocol control provides an interface for requesting registration information for an Internet domain name. When a domain name is registered, the organization that registers the domain must provide certain contact information along with technical information such as the primary name servers for that domain. The control provides an interface for requesting that information and returning it to the program so that it can be displayed or processed. This control would be most commonly used to query the Whois server at **whois.internic.net** to obtain information about a specific Internet domain name or an administrative contact at that domain.

Requirements

The SocketTools ActiveX Edition components are self-registering controls compatible with any programming language that supports COM (Component Object Model) and the ActiveX control specification. If you are using Visual Basic 6.0 you must have Service Pack 6 (SP6) installed. It is recommended that you install all updates for your development tools.

This control is supported on Windows 7, Windows Server 2008 R2 and later versions of the desktop and server platforms. If you are using Windows 7, you must have Service Pack 1 (SP1) installed as a minimum requirement. It is recommended that you install the current service pack and all critical updates available for the operating system.

This product includes both 32-bit and 64-bit ActiveX controls. Native 64-bit CPU support requires the latest 64-bit version of Windows 7, Windows Server 2008 R2 or later versions of the Windows operating system.

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.

Whois Protocol Control Properties

Property	Description
AutoResolve	Determines if host names and IP addresses are automatically resolved
Blocking	Gets and sets the blocking state of the control
HostAddress	Gets and sets the IP address of the server
HostName	Gets and sets the name of the server
IsBlocked	Return if the control is blocked performing an operation
IsConnected	Determine if the control is connected to a server
IsInitialized	Determine if the control has been initialized
IsReadable	Return if data can be read from the server without blocking
Keyword	Specify the keyword to search for
LastError	Gets and sets the last error that occurred on the control
LastErrorString	Return a description of the last error to occur
RemotePort	Gets and sets the port number for a remote connection
SearchType	Specify the type of query to be performed by the server
ThrowError	Enable or disable error handling by the container of the control
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 object

AutoResolve Property

Determines if host names and IP 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 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 **AutoResolve** to False.

Data Type

Boolean

See Also

HostAddress Property, HostName Property

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 read data when none has been written, an error is generated. Events such as **OnConnect**, **OnDisconnect**, **OnRead** and **OnWrite** are only fired if the connection is non-blocking.

Data Type

Boolean

See Also

IsBlocked Property, IsReadable Property

HostAddress Property

Gets and sets the IP address of the server.

Syntax

object.**HostAddress** [= *ipaddress*]

Remarks

The **HostAddress** property can be used to set the IP address for a server 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, HostName Property

HostName Property

Gets and sets the name of the server.

Syntax

object.**HostName** [= *hostname*]

Remarks

The **HostName** property should be set to the name of the server 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

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.

Note that this property will return True if there is ***any*** blocking operation being performed by the application, regardless if the specified control is responsible for the blocking operation or not.

Data Type

Boolean

See Also

Blocking Property, LastError Property

IsConnected Property

Determine if the control is connected to a server.

Syntax

object.**IsConnected**

Remarks

The **IsConnected** read-only property is set to a value of true if the control is connected with a server, otherwise the property has a value of false.

Data Type

Boolean

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

IsReadable Property

Return if data can be read from the server without blocking.

Syntax

object.**IsReadable**

Remarks

The **IsReadable** property returns True if data can be read from the server without blocking. For non-blocking connections, this property can be checked before the application attempts to read the data, preventing an error.

Data Type

Boolean

See Also

IsConnected Property, Read Method, OnRead Event

Keyword Property

Specify the keyword to search for.

Syntax

object.**Keyword** [= *value*]

Remarks

The **Keyword** property specifies the value used when querying the server. The keyword may refer to a handle, a user name or a mailbox name. Setting this property provides the default keyword for the **Search** method.

Keywords may contain special characters that instruct the server how to match the value. These values are outlined in RFC 954, the standards document that describes the WHOIS/NICNAME protocol. These forms are typically recognized:

Example	Description
value	Search for value as either a user name or a handle
value...	Search for value that matches anything up to that point
!value	Search for a handle that matches the given value
last, first	Search for the specified name
user@	Search for mailboxes with the given user name
@host	Search for mailboxes on the specified host
user@host	Search for mailboxes for the user on the specified host

If the keyword uses any of these special forms, the **SearchType** property must be set to **whoisSearchAny**, which tells the control not to modify the keyword value when submitting the query to the server. Note that all keyword forms may not be supported by a given server, and additional types of searches may be supported.

Data Type

String

See Also

SearchType Property, Connect Method, Search 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, OnError Event

LastErrorString Property

Return a description of the last error to occur.

Syntax

object.**LastErrorString**

Remarks

The **LastErrorString** property returns a description of the last error that occurred. This can be used to display a meaningful error message to a user, rather than just the numeric value returned by the **LastError** property.

Data Type

String

See Also

LastError Property, OnError Event

RemotePort Property

Gets and sets the port number for a remote connection.

Syntax

object.**RemotePort** [= *portnumber*]

Remarks

The **RemotePort** property is used to set the port number that the control will use to establish a connection with the server.

Data Type

Integer (Int32)

See Also

HostAddress Property, HostName Property

SearchType Property

Specify the type of query to be performed by the server.

Syntax

object.**SearchType** [= *value*]

Remarks

The **SearchType** property specifies the default type of query to be performed by the server using the **Search** method. The following table lists the types of searches that may be performed:

Value	Constant	Description
whoisSearchAny	Search for any value that matches the given keyword value	
whoisSearchHandle	Search for a handle that matches the given keyword value	
whoisSearchName	Search for a user name that matches the given keyword value	
whoisSearchMailbox	Search for a user mailbox that matches the given keyword value	

If you wish to perform a more complex query using the syntax outlined in RFC 954, specify a search type of **whoisSearchAny** and then provide the search string value that you want to submit.

Data Type

Integer (Int32)

See Also

Keyword Property, Connect Method, Search Method

ThrowError Property

Enable or disable error handling by the container of the control.

Syntax

object.**ThrowError** = { True | False }

Remarks

Error handling for 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.

If the **ThrowError** property is set to True, then errors occurring within the control will be thrown to the container of the control. In addition, the **OnError** event will fire. For example, in Visual Basic, it is recommended that the **OnError** mechanism be used to catch errors.

Note that if an error occurs while a property value is being accessed, an error will be raised regardless of the value of the **ThrowError** property, but the **OnError** event will not be fired.

Data Type

Boolean

Example

The following example handles errors by checking the return code of a method:

```
WhoisClient1.ThrowError = False
nError = WhoisClient1.Connect(strHostName)

If nError > 0 Then
    MsgBox WhoisClient1.LastErrorString, vbExclamation
    Exit Sub
Endif
```

The following example handles errors by throwing them to the container:

```
On Error Resume Next: Err.Clear

WhoisClient1.ThrowError = True
WhoisClient1.Connect strHostName

If Err.Number <> 0
    MsgBox Err.Description, vbExclamation
    Exit Sub
Endif
On Error GoTo 0
```

See Also

LastError Property, LastErrorString Property, OnError Event

Timeout Property

Gets and sets the amount of time until a blocking operation fails.

Syntax

object.**Timeout** [= *seconds*]

Remarks

Setting the **Timeout** property specifies the number of seconds until a blocking operation fails and the control returns an error.

Note that the **Timeout** property also determines the amount of time the control will spend attempting to connect to a server. If a connection is not established within the given time period, the connection attempt will fail.

Data Type

Integer (Int32)

Trace Property

Enable or disable socket function level tracing.

Syntax

object.**Trace** [= { True | False }]

Remarks

The **Trace** property is used to enable or disable the logging of Windows Sockets 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 networking controls, and is enabled or disabled for an entire process. This means that once tracing is enabled for a given control, all of the function calls made by the process using any of the SocketTools controls will be logged. For example, if you have an application using both the FTP and POP3 controls, and you set the **Trace** property to True on the FTP control, function calls made by both the FTP and POP3 controls will be logged. Additionally, enabling a trace is cumulative, and tracing is not stopped until it is disabled for all controls used by the process.

If tracing 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 trace file is fairly large. 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.

Note that only those function calls made by the SocketTools networking controls will be logged. Calls made directly to the Windows Sockets API, or calls made by other controls, will not be logged.

Data Type

Boolean

See Also

TraceFile Property, TraceFlags Property

TraceFile Property

Specify the socket function trace output file.

Syntax

object.**TraceFile** [= *filename*]

Remarks

The **TraceFile** property is used to specify the name of the trace file that is created when socket function tracing is enabled. If this property is set to an empty string (the default value), then a file named **cstrace.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 socket function tracing is enabled per-process, the trace file is shared by all instances of the controls being used. If multiple controls have tracing enabled, the **TraceFile** property should be set to the same value for each control. 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:

```
VB6 105020 0000 INF: WSAAsyncSelect(46, 0xcc4, 0x7e9, 0x27) returned 0
VB6 105020 0015 WRN: connect(46, 192.0.0.1:1234, 16) returned -1 [10035]
VB6 111535 0000 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 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 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.

Data Type

String

See Also

Trace Property, TraceFlags Property

TraceFlags Property

Gets and sets the socket function tracing flags.

Syntax

object.**TraceFlags** [= *traceflags*]

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
whoisTraceInfo	All function calls are written to the trace file, including information about successful calls made to the networking library. This is the default value.
whoisTraceError	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.
whoisTraceWarning	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.
whoisTraceHexDump	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 server.

Since function logging is enabled per-process, the trace flags are shared by all instances of the controls 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 controls 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 object.

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

Whois Protocol Control Methods

Method	Description
Cancel	Cancels the current blocking network operation
Connect	Establish a connection with a server
Disconnect	Terminate the connection with a server
Initialize	Initialize the control and validate the runtime license key
Read	Return data read from the server
Reset	Reset the internal state of the control
Search	Search for the specified record
Uninitialize	Uninitialize the control and release any system resources that were allocated

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

Disconnect Method, Reset Method, OnCancel Event

Connect Method

Establish a connection with a server.

Syntax

object.**Connect**([*RemoteHost*], [*RemotePort*], [*Timeout*], [*Options*])

Parameters

RemoteHost

A string which specifies the host name or IP address of the server. If this argument is not specified, it defaults to the value of the **HostAddress** property if it is defined. Otherwise, it defaults to the value of the **HostName** property.

RemotePort

A number which specifies the port to connect to on the server. If this argument is not specified, it defaults to the value of the **RemotePort** property. A value of zero indicates that the default port number for this service should be used to establish the connection.

Timeout

The number of seconds that the client will wait for a response before failing the operation. If this argument is not specified, the value of the **Timeout** property will be used as the default.

Options

A reserved parameter. This argument should either be omitted, or always be zero.

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

HostAddress Property, HostName Property, RemotePort Property, Disconnect Method, OnConnect Event

Disconnect Method

Terminate the connection with a server.

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

IsConnected Property, Connect Method, OnDisconnect Event

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 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 whoisClient = CreateObject("SocketTools.WhoisClient.11")

nError = whoisClient.Initialize(strLicenseKey)
If nError > 0 Then
    MsgBox "Unable to initialize SocketTools component"
    End
End If
```

See Also

IsInitialized Property, Uninitialize Method

Read Method

Return data read from the server.

Syntax

object.**Read**(*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

The number of bytes actually read from the server 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 server, 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.

See Also

IsConnected Property, IsReadable Property, OnRead 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

Search Method

Search for a record using the specified keyword.

Syntax

object.**Search**([*Keyword*], [*SearchType*])

Parameters

Keyword

An optional string which specifies the keyword to search for. Typically this is the name of a domain, user handle or an email address. If this argument is omitted, the value of the **Keyword** property is used as the default value.

SearchType

An optional integer value which specifies the type of search to perform. If this argument is omitted, the value of the **SearchType** property is used as the default value. One of the following search types may be specified:

Value	Description
whoisSearchAny	Search for any value that matches the given keyword value
whoisSearchHandle	Search for a handle that matches the given keyword value
whoisSearchName	Search for a user name that matches the given keyword value
whoisSearchMailbox	Search for a user mailbox that matches the given keyword value

Return Value

A value of zero is returned if the method succeeds. Otherwise, a non-zero error code is returned which indicates the cause of the failure.

Remarks

The **Search** method submits the specified keyword and type of search to the server. The data returned by the server can be read using the **Read** method. Note that the text returned by a UNIX based server may only contain linefeeds at the end of each line of text, rather than the standard carriage return/linefeed used on Windows systems.

Example

The following example demonstrates how to use the **Search** method:

```
Dim strBuffer As String, strResults As String
Dim nRead As Long, nError As Long

' Connect to the InterNIC server and submit a search for the
' Internet Engineering Task Force (IETF) domain
nError = WhoisClient1.Connect("whois.internic.net")
If nError = 0 Then
   nError = WhoisClient1.Search("ietf.org", whoisSearchAny)
End If

' If an error occurs, display a message box and exit
If nError > 0 Then
    MsgBox WhoisClient1.LastErrorString, vbExclamation
    Exit Sub
End If
```

```
' Read the data returned by the server and store it in
' a string buffer named strResults
Do
    nRead = WhoisClient1.Read(strBuffer, 2048)
    If nRead < 1 Then Exit Do
    strResults = strResults + strBuffer
Loop

' If there was an error reading the data, then report
' it to the user
If nRead = -1 Then
    MsgBox WhoisClient1.LastErrorString, vbExclamation
End If

' Disconnect from the server
WhoisClient1.Disconnect
```

See Also

Keyword Property, SearchType Property, Connect Method, Read 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. 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

Whois Protocol Control Events

Event	Description
OnCancel	This event is generated when a blocking operation is canceled
OnConnect	This event is generated when a connection is established
OnDisconnect	This event is generated when a connection is terminated
OnError	This event is generated when a control error occurs
OnRead	This event is generated when data is available to be read
OnTimeout	This event is generated when a blocking operation times out

OnCancel Event

The **OnCancel** event is generated when a blocking operation is canceled.

Syntax

Sub *object*_**OnCancel** ([*Index* **As Integer**])

Remarks

This event is generated 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, OnTimeout 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 server as a result of a **Connect** method call. This event is only triggered when the **Blocking** property is set to False.

See Also

Blocking Property, 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 server. This event is only triggered when the **Blocking** property is set to False.

When the **OnDisconnect** event fires, it is possible that there may still be buffered data available to read from the server. Before disconnecting from the server, the application should attempt to read any remaining data until the **Read** method returns a value of zero, or returns an error indicating that the operation would block.

See Also

Blocking Property, IsConnected Property, IsReadable Property, Connect Method, Disconnect Method, Read Method, 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 is generated when an error occurs during a control action. Errors not generated by the control itself, such as errors related to the programming language or general component errors, do not trigger 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

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 triggered when the **Blocking** property is set to False.

See Also

IsReadable Property, Read Method, Search Method

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. This event is only triggered when the **Blocking** property is set to True.

See Also

Timeout Property, OnCancel Event