WinSock Networking VBX v1.08*alpha* HELP

When you add this VBX to your project, three new custom controls will be added to your Toolbox. You can select the appropriate control by clicking the corresponding button.

See About this project for more general information
See the Changes that have been made to this version.
See the public domain Copyright information for distribution availability.
See the Common Problems information for general help on other issues.

The toolbox buttons available through WSANET.VBX are:

 NetClient control  NetServer control  Ini control

Domain **N**ame **S**ystem
Fully **Q**ualified **D**omain **N**ame

Berkley **S**oftware **D**istribution

First **I**n **F**irst **O**ut

About the WinSock Networking VBX Project

About the NetClient Control

This control was written between July and December of 1993 as a side project to other Windows Sockets applications. Somehow, between 3 harddrive crashes, this project managed to survive to the stage you see it in today. NetClient's origional purpose was to fill the gap between the many Unix networking services and the moderate few WinSock applications.

About the Author

(As of October of 1993)(In an attempt at modest second person tone)

Ian Blenke is currently a full time student teaching himself the wonders of computing. He currently takes part in Windows Sockets discussions through Internet mail, and tries to read what he can through Usenet news. He has migrated from his developing stage on the Commodore 64 throughout high school as a "demo" programmer to the more advanced Unix and Windows paradigms in todays workplace. He currently has currently finished his initial 2 years at a local community college, and hopes to continue his education with either the Florida Institute of Technology (FIT) or the University of Central Florida (UCF) in their Software Engineering programs.

Ian currently works as a COOP for Harris Corporation Corporate Headquarters in Melbourne, Florida. As a COOP he is kept busy putting PCs together, installing software, and occationally writing miscellaneous "hacks" to do such things as monitor the network with batch files and VB.

Ian programs during his spare evenings at home. Since his college studies have began, he has devoted his personal software to the public domain. In the future he hopes to continue writing network and GUI applications for whatever OS has his eye at the time. As time progresses, he hopes to continue to contribute to the new "Electronic Superhighway" (TM Gore) (I.E. Internet) as it expands into its full glory.

E-Mail addresses:

iblenke@ic1d.harris.com
iblenke@rhino.ess.harris.com

Thank you's and Handshakes

This section is devoted to you. Whenever I receive a good idea, or a successful nudge to implement something new or fix an outstanding bug, I will try and remember and put your name in here. If I don't, simply remind me, and I will gladly put you in (because I realize that I have a very short memory ;).

For the idea of a WinSock control at all, I would like to thank G. Michael Carr, a co-worker. If it weren't for his daily trips to bug me about finishing it, you would likely have never read this!

For the idea of RSH/LPD type restricted port clients, I would like to thank Dan Tenenbaum at Microsoft. This was on my agenda - I just didn't feel like implementing it, but his persistance gave me a reason to implement the LocalPort property. He also aided the project by creating a VC++ makefile, reporting a whole slew of warnings from the VC++ compiler (which, hopefully, have been fixed), and noting that WEP was unnecessarily exported in the WSNETC.DEF file (a holdover from an earlier revision).

For relaying constant feedback as to the status of his Sequoia program, I would like to thank Jason Levine at Columbia University. Jason has spent an incredible amount of time getting his Sequoia program to work with WSMTPD and the WSANET VBX. I have to thank you for the amount of time you've put into this wonderful mail program.

For testing and reporting bugs at the WinSock level over Peter Tattam's WinSock, I would like to thank Mike Rogers at the University of Iowa. His comments relate both to the real world problems that might occur with the NetClient control, and to the problems that users will face with my VB3.0 only examples (which HAVE been rewritten for VB2.0 standard).

For testing the Block/SendBlock/RecvBlock properties with his Gopher client, I would like to thank Rod Potter from York University (Computing & Communications Services). By looking at his CNGopher source code, I have corrected some of the online help documentation by seeing what everyone expects the VBX to do.

For suggesting the Host property (for the heck of it), and for giving me ideas for UDP, OOB, and Raw socket support in the future, I would like to thank Tom Hogard from the Air Force Logistics Command.

Common Problems with WSANET.VBX (FAQ)

Q. *Whenever I use WSANET.VBX in my project, my machine slows to a crawl!*

A. This problem is due to the fact that you are using the debugging version of the VBX. In order to remedy this problem, you MUST run DBWIN.EXE or OX.SYS in order to install the debugging "hook" that Windows keeps looking for during each OutputDebugString() call. The other solution is to use RELEASE.VBX by renaming it to WSANET.VBX - it uses no such debugging calls - and it will definitely run faster. Due to the massive output of the debugging version, you might consider using the RELEASE.VBX version (by copying it to your SYSTEM directory as WSANET.VBX) until you stumble upon a bug.

Q. *Whenever I try to send a large file, I get wierd results! What gives?*

A. This is a side-effect of asyncronous programming. In order to send more data than can be held in your WINSOCK.DLL's buffers, and NetClient's send buffers, you MUST either poll SendCount or use the SendThreshold/OnSend() pair. See the WFingerD or VBSMTPD example of how to do exactly this.

Q. *Your example programs are written explicitly for VB2.0 and newer! What about VB1.0 users?*

A. Unfortunately, the author does not have VB1.0 or VC++ (MFC 2.0) to test out or write sample applications. He has, however, tried to make the text-saved VB2.0 examples generic enough for VB1.0 use. You merely have to load an example form into a text editor (like Notepad) and use the Copy function - then Paste what you copied into a VB1.0 form.

Q. *What are your plans for this VBX?*

A. WSANET.VBX is intended to be used by Public Domain application authors as a FREEWARE tool. This entire project is merely a pet project of mine to get TCP/IP applications quickly to the end-users - as FREEWARE. The Windows Socket v1.1 standard was a feat in itself, but the only real world applications that I have seen on Internet are Shareware packages and commercial demos. This tool will make EVERY client/server application available on Unix to be written for Windows 3.x and newer (as long as a WINSOCK.DLL thunk is included). The main reason for this control, trying not to sound ego-centered, was to show that I have some potential as a future Software Engineer.

What is a Socket?

The concept of Sockets is derived from the network programming model on BSD Unix. A socket was origionally intended as a way for two machines to talk across the network without the hassle of packet derived communication.

A Socket is (at least on NT and Unix) a unique "handle" that can be used on the system to reference the connection to another host. A Socket has 2 sides to it: Local and Remote. The Local side uses your PC's TCP/IP address (actually INADDR_ANY for WinSock calls) and the LocalPort property to define the application's side of the connection. By setting HostAddr (I.P.) or HostName (DNS) and the RemotePort to a remote host's TCP address, you tell the NetClient control where to connect to.

By using this control, the NetClient control itself becomes the Socket. You can set the LocalPort and RemotePort properties to allow the control to bind() both sides of a real WinSock socket appropriately. When you set the HostAddr or HostName property, you are telling the control that it is to try and connect to that address. Once you set Connect to True, you allow the control to allocate a real WinSock socket to attempt a connection to the remote host. If you use the Index property, you can have multiple Windows Sockets client connections active simultaneously. Each control in the control array receives its own WinSock "environment" that you can use to communicate across the network.

WinSock Errors

The following is a list of possible error codes returned by the NetClient control, along with their explanations. The error numbers are consistently set across all Windows Sockets-compliant implementations.

Windows Sockets code	Berkeley equivalent	Error	Interpretation
WSAEINTR	EINTR	10004	As in standard C
WSAEBADF	EBADF	10009	As in standard C
WSEACCES	EACCES	10013	As in standard C
WSAEFAULT	EFAULT	10014	As in standard C
WSAEINVAL	EINVAL	10022	As in standard C
WSAEMFILE	EMFILE	10024	As in standard C
WSAEWOULDBLOCK	EWOULDBLOCK	10035	As in BSD
WSAEINPROGRESS	EINPROGRESS	10036	This error is returned if anyWindows Sockets API function is called while a blocking function is in progress.
WSAEALREADY	EALREADY	10037	As in BSD
WSAENOTSOCK	ENOTSOCK	10038	As in BSD
WSAEDESTADDRREQ	EDESTADDRREQ	10039	As in BSD
WSAEMSGSIZE	EMSGSIZE	10040	As in BSD
WSAEPROTOTYPE	EPROTOTYPE	10041	As in BSD
WSAENOPROTOOPT	ENOPROTOOPT	10042	As in BSD
WSAEPROTONOSUPPORT	EPROTONOSUPPORT	10043	As in BSD
WSAESOCKTNOSUPPORT	ESOCKTNOSUPPORT	10044	As in BSD
WSAEOPNOTSUPP	EOPNOTSUPP	10045	As in BSD
WSAEPFNOSUPPORT	EPFNOSUPPORT	10046	As in BSD
WSAEAFNOSUPPORT	EAFNOSUPPORT	10047	As in BSD
WSAEADDRINUSE	EADDRINUSE	10048	As in BSD
WSAEADDRNOTAVAIL	EADDRNOTAVAIL	10049	As in BSD
WSAENETDOWN	ENETDOWN	10050	As in BSD. This error may be reported at any time if the Windows Sockets implementation detects an underlying failure.
WSAENETUNREACH	ENETUNREACH	10051	As in BSD
WSAENETRESET	ENETRESET	10052	As in BSD
WSAECONNABORTED	ECONNABORTED	10053	As in BSD
WSAECONNRESET	ECONNRESET	10054	As in BSD
WSAENOBUFS	ENOBUFS	10055	As in BSD
WSAEISCONN	EISCONN	10056	As in BSD
WSAENOTCONN	ENOTCONN	10057	As in BSD
WSAESHUTDOWN	ESHUTDOWN	10058	As in BSD
WSAETOOMANYREFS	ETOOMANYREFS	10059	As in BSD
WSAETIMEDOUT	ETIMEDOUT	10060	As in BSD
WSAECONNREFUSED	ECONNREFUSED	10061	As in BSD
WSAELOOP	ELOOP	10062	As in BSD
WSAENAMETOOLONG	ENAMETOOLONG	10063	As in BSD
WSAEHOSTDOWN	EHOSTDOWN	10064	As in BSD
WSAEHOSTUNREACH	EHOSTUNREACH	10065	As in BSD
WSASYSNOTREADY		10091	Indicates that the network subsystem is unusable.
WSAVERNOTSUPPORTED		10092	Indicates that the Windows Sockets DLL cannot support this app.
WSANOTINITIALISED		10093	Indicates that a successful **WSAStartup()** has not yet been performed.
WSAHOST_NOT_FOUND	HOST_NOT_FOUND	11001	As in BSD.
WSATRY_AGAIN	TRY_AGAIN	11002	As in BSD
WSANO_RECOVERY	NO_RECOVERY	11003	As in BSD
WSANO_DATA	NO_DATA	11004	As in BSD

The first set of definitions is present to resolve contentions between standard C error codes which may be defined inconsistently between various C compilers.

The second set of definitions provides Windows Sockets versions of regular Berkeley Sockets error codes.

The third set of definitions consists of extended Windows Sockets-specific error codes.

The fourth set of errors are returned by Windows Sockets gethostbyXXXX() calls, and correspond to the errors which in Berkeley software would be returned in the *h_errno* variable. They correspond to various failures which may be returned by the Domain Name Service. If the Windows Sockets implementation does not use the DNS, it will use the most appropriate code. In general, a Windows Sockets application should interpret WSAHOST_NOT_FOUND and WSANO_DATA as indicating that the key (name, address, etc.) was not found,, while WSATRY_AGAIN and WSANO_RECOVERY suggest that the name service itself is non-operational.

The error numbers are derived from the **winsock.h** header file, and are based on the fact that Windows Sockets error numbers are computed by adding 10000 to the "normal" Berkeley error number.

Note that this table does not include all of the error codes defined in **winsock.h**. This is because it includes only errors which might reasonably be returned by a Windows Sockets implementation: **winsock.h**, on the other hand, includes a full set of BSD definitions to ensure compatibility with ported software.

This Error section was "stolen" from the WinSock.HLP source file (winsockx.rtf, part of the Microdyne.com archives). You can get the actual WinSock.HLP file from a number of various FTP sites, along with the .H/.DEF/.LIB files and sample C sources.

Copyright

In this document, **VBX** refers to the actual Visual Basic control distributed with and based on this WSANET project, and **author** refers to **Ian Blenke**. The **author** cannot be held responsible for damages, express or implied, for the use of this software. No commercial use can be made of this product without the consent of the **author**. No profit of any kind can be made on the sale or distribution of this program. If you wish to distribute this program with other samples of WinSock programming, you must first contact the **author** so that he can keep accurate records of its usage; no charge may be made for this project's availability other than the cost of the physical medium used to store it on and any processing costs. If you write any programs based on this source code, including parts of this source code, or in some way derived from this source code, you may not sell them for any profit without the written consent of the **author**. If you incorporate this VBX into a *free* public domain program, all the **author** requires is a notification that "The WinSock VBX control was written by **Ian Blenke**" and some form of notification that his name was used in the public domain software distribution. No profit of any kind, shareware/commercial or otherwise, may be made for software based on this **VBX** without the written consent of the **author**. This does not represent a contract on the part of the **author**. *If any issues cannot clearly be resolved by reading this text, immediately contact the* **author**.

The **VBX** control distributed with this project must be distributed with your programs free of charge. You cannot charge money for any program based on the **VBX** without the written consent of the **author**.

Notes:

If you have any bug reports, by all means, tell me! I would be glad to help keep this code up to date.

The source of the project is included in the open spirit under which most Unix source is distributed: if you modify it in some unique way, PLEASE change the LIBRARY name in WSANET.DEF to match the filename of the finished VBX. This is to make sure that other WSANET applications continue to work as their designers intended. If you make any modifications, please contact the author - your "hacks" could likely be incorporated into the next release of WSANET.VBX and made available to other developers.

I don't like such agreements. But in today's world of lawyers and lawbreakers, I have little other choice. Enjoy!

Changes

v1.03alpha to v1.04alpha

Online context sensitive help is stable and complete (for this version)

v1.04alpha to v1.05alpha

Socket semantics have changed (although the ability to set it is not yet technically implemented).

New Toolbox BMP has been designed:


New (extended from SOCKET.ICO)


Old (drawn by author, no artistic value ;)

HLP updated: **Changes** topic added.

The LocalPort property was broken previous to version 1.05alpha. Setting it would set the RemotePort property.

WSNETC.C and NETC.C now use the DEBUG_BUILD flag to mark debug/release build versions (instead of keeping two copies of source).

WSNETC.C cleaned up. Debugging information clearer now.

NETC.C cleaned up. Debugging information clearer now.

NET.C - Added "netConnect()" procedure to aid in the setting of the Socket property

All sources cleaned up for compilation by VC++ (uncomment USE_VCPP in WSNetC.H)

Fixed TimeOut bug. Previous versions will ALWAYS delay for just 1 second.

v1.05alpha to v1.06alpha

Added Version and Debug properties.

Added HostAliasCount/HostAliasList() and HostAddressCount/HostAddressList() properties

Changed the HostAddr/HostName property code to avoid VB calls where possible.

Added Ini Control - 6 properties, no events

Changed the names of EVERYTHING

Moved everything around

Caused general havok within the code:

Init.C is renamed to WSANet.C, and 2 procedures from WSNetC.C were added.
WSNetC.C is copied over to NetClnt.C
Ini.C/Ini.H is added (to support the Ini control)
WSNetC.H becomes two files: NetClnt.H and WSANet.H
Functions added to Net.C: GetAtomHsz(), SetAtom()

The project name has changed from WSNETC.VBX to WSANET.VBX

Reason for changes:

NetClnt.C/NetClnt.H now hold all of the NetClient control code
Ini.C/Ini.H now hold all of the Ini control code
WSANet.C/WSANet.H is the "glue" for the project.
Version.H holds all version information
WSANet.RC/WSANet.DEF hold the resources and definitions

Caused the programmer many headaches as he renamed everything ;)

Fixed WFinger/WSMTPC/VBTest to run under VB2.0 standard.

Added fully commented WFinger example code to the HLP file.

v1.06alpha to v1.07alpha

Added the NetServer control.

Fixed a nasty problem with v1.06alpha's use of ATOMs to store strings: all strings are now stored internally as Visual Basic strings (type HLSTR).

Changed the semantics of setting the Socket on the NetClient control so that NetServer's OnAccept(Socket, ...) could be handed over.

Fixed a potential problem with the FD_CLOSE message handler: it didn't "zero out" the Socket when it closed!

Dropped the VBTest client due to it's obvious inadequacies (and pointlessness)
Wrote WinWhois client.
Wrote WFingerD server.
Modified all VB3.0 examples to use a "standard" GUI appearance.
Shuffled the HLP file around. Topics have changed to include the control name along with the property context string.

v1.07alpha to v1.08alpha

Fixed a problem that caused WSAE_NOCONN errors.
Fixed a event firing problem.
Fixed the connection semantics
Added window position "memory" to VB3.0 samples
Wrote VB-SMTPD as a multi-connection state server which uses dynamic creation and destruction of NetClient controls.
Removed example code from the HLP file: the SAMPLES directory holds all source code, why keep 2 copies up to date?
Setting Listen to LISTEN_DONTREUSE (2) will not open a server socket as SO_REUSEADDR.
Added the Host property.
Fixed a Line property bug. Not only was it poorly coded before, I wonder what I was thinking at the time.

To-Do list:

- Make sure the binary transfer properties and functions are working correctly.
- Make VC++ "sample" applications
- Add OnGetHost() event, and AsyncHandle property (for non-blocking DNS lookups)
- Add TCP/UDP and OOB switch.

Unused Property

You, somehow, have managed to start up an older version of this HLP file with a newer version of WSANET.VBX. The context-sensitive help that you have selected is for a control property or event that this HLP file does not know about. Please check your system path for the older HLP file, and replace it with the new one.

Have fun!

Ini Management Control

Properties Methods Events Errors Examples

Description

You can use the **Ini** control to access the Windows API calls that read to and write from .INI files. This control is invisible at run time, much like the timer control provided with Visual Basic.

File name

WSANET.VBX

Object Type

Ini

Remarks

This control encapsulates the standard Windows API calls WritePrivateProfileString() and GetPrivateProfileString(). The **Ini** control makes it possible to read to and write from .INI files with little programming effort. The rationale behind including this control in the WSANET project was twofold; The **Ini** control is small, and almost every TCP/IP application requires some form of database or state information.

INI Format

```
Filename:
  [Section]
  Entry = Value
```

Properties

These are the properties for the **Ini** Control. Properties that apply *only* to this control, or that require special consideration when using it, are marked with an asterisk (*).

CtlName	Index	Tag
*Default	Left	Top
*Entry	Parent	*Value
*Filename	*Section	

Value is the default property

Methods

This current version of the Ini control supports no custom Methods.

Events

This current version of the Ini control includes no standard or custom Events.

Errors

This current version of the Ini control will ONLY return one error:

ERR_OUTOFMEMORY - Error #7 - Out of memory

Examples

Here is a quick example of how to use the Ini control. The following is a fictitious .INI file:

```
{sequoia.ini}
  [Sequoia]
  MailFile=C:\MAIL\MAIL.TXT
  CheckInterval=30
  FullName=Ian C. Blenke
  HomeAddress=iblenke@rhino.ess.harris.com
  SMTPHostIP=130.41.1.251
  TempDir=C:\WINDOWS\TEMP
```

Here are two quick subroutines that can access any entry in the above file:

```
Function GetSequoiaEntry(sEntry As String, sDefault As String) As String
  Ini.Filename = "sequoia.ini"
  Ini.Section = "Sequoia"
  Ini.Entry = sEntry
  Ini.Default = sDefault
  GetSequoiaEntry = Ini.Value
End Function

Sub SetSequoiaEntry(sEntry As String, sValue As String)
  Ini.Filename = "sequoia.ini"
  Ini.Section = "Sequoia"
  Ini.Entry = sEntry
  Ini.Value = sValue
End Sub
```

Usage

From here, you could set the HomeAddress entry to "iblenke@ic1d.harris.com" with the code:

```
SetSequoiaEntry("HomeAddress", "iblenke@ic1d.harris.com")
```

and you could read the HomeAddress entry with:

```
sSavedAddress = GetSequoiaEntry("HomeAddress", "")
```

YES, It really *is* that easy!

Note: You DO NOT need to set the properties in any given order. As long as the appropriate properties are set before you use the Value property, you do not need to set them every time (they are stored internally as VB strings for each instance of the INI control).

Default Property, Ini Control

Description

This property is the **Default** value returned through the Value property if no Entry in [Section] of a Filename can be found.

Visual Basic

[*form .*]*Ini* .**Default**[= *value$*]
[*value$*] = [*form .*] *Ini*.**Default**

Visual C++

pIni->**GetStrProperty("Default")**
pIni->**SetStrProperty("Default",** *eol$***)**

Remarks

This property corresponds to the Default argument to a GetPrivateProfileString() call.

VB Errors

ERR_OUTOFMEMORY (7)	-	When **Default** is read, and a response string could not be created.

Data Type

String (HSZ)

Scope

VB1.0+/VC++

Entry Property, Ini Control

Description

This property is the **Entry** line in the [Section] of a Filename that you must set in order to use the Value property.

Visual Basic

[*form .*]*Ini* .**Entry**[= *entry$*]
[*entry$*] = [*form .*] *Ini*.**Entry**

Visual C++

pIni->**GetStrProperty("Entry")**
pIni->**SetStrProperty("Entry",** *entry$***)**

Remarks

This property corresponds to the Entry argument given to both the GetPrivateProfileString() and WritePrivateProfileString() calls.

If the Entry in a [Section] of the .INI file does not exist, when Value is set, the Entry will be created.

If the Entry in a [Section] in the .INI file does not exist, when Value is read, the Default value is returned.

VB Errors

ERR_OUTOFMEMORY (7) - When **Entry** is read, and a response string could not be created.

Data Type

String (HSZ)

Scope

VB1.0+/VC++

Filename Property, Ini Control

Description

This property is the **Filename** of the .INI file from which to extract a Value from a Entry in a [Section]

Visual Basic

[*form .*]*Ini* .**Filename**[= *ini_filename$*]
[*ini_filename$*] = [*form .*] *Ini*.**Filename**

Visual C++

pIni->**GetStrProperty("Filename")**
pIni->**SetStrProperty("Filename",** *ini_filename$***)**

Remarks

This property corresponds to the **Filename** argument given to both the GetPrivateProfileString() and WritePrivateProfileString() calls. You do NOT have to use a full path name: in fact, it is better to use just the base .INI filename in order to allow users to keep their .INI files wherever they wish in their path. The Windows API calls take care of the underlying searches, you merely have to set this property to the name of an .INI file somewhere in the system path.

If the .INI file corresponding to **Filename** does not exist, when Value is set, the .INI file will be created.

If the .INI file corresponding to **Filename** does not exist, when Value is read, the Default value is returned.

VB Errors

ERR_OUTOFMEMORY (7) - When **Filename** is read, and a response string could not be created.

Data Type

String (HSZ)

Scope

VB1.0+/VC++

Section Property, Ini Control

Description

This property is the Section in a .INI Filename from which to extract a Value from a Entry.

Visual Basic

[*form .*]*Ini* .**Section**[= *section$*]
[*section$*] = [*form .*] *Ini*.**Section**

Visual C++

pIni->**GetStrProperty("Section")**
pIni->**SetStrProperty("Section",** *section$***)**

Remarks

This property corresponds to the **Section** argument given to both the GetPrivateProfileString() and WritePrivateProfileString() calls.

If the [Section] in an .INI file corresponding to the **Section** property does not exist, when Value is set, the [Section] will be created.

If the [Section] in an .INI file corresponding to the **Section** property does not exist, when Value is read, the Default value is returned.

VB Errors

ERR_OUTOFMEMORY (7) - When **Section** is read, and a response string could not be created.

Data Type

String (HSZ)

Scope

VB1.0+/VC++

Value Property, Ini Control

Description

This property is the actual **Value** that is set as an Entry in a Section in an .INI Filename. This property is the default property of the **Ini** control.

Visual Basic

[*form .*]*Ini* .**Value**[= *value$*]
[*value$*] = [*form .*] *Ini*.**Value**

Visual C++

pIni->**GetStrProperty("Value")**
pIni->**SetStrProperty("Value",** *value$***)**

Remarks

This property corresponds to the actual buffer or string argument given to both the GetPrivateProfileString() and WritePrivateProfileString() calls.

If the Filename, Section, or Entry do not exist, the **Value** property will return what is in the Default property when read.

If the Filename, Section, or Entry do not exist, and the **Value** property is set, the appropriate fields in the .INI file will be created.

VB Errors

ERR_OUTOFMEMORY (7) - When **Value** is read, and a response string could not be created.

Data Type

String (HSZ)

Scope

VB1.0+/VC++

WinSock Network Server Control

Properties Methods Events Errors Examples

Description

You can use the NetServer control to access the Windows Sockets v1.1 server calls. This control is invisible at run time, much like the timer control provided with Visual Basic.

File name

WSANET.VBX

Object Type

NetServer

Remarks

The **NetServer** control uses functionality provided by a vendor specific version of WINSOCK.DLL that is provided with your TCP/IP protocol kernel. With this control, you can export any TCP service that you want for other TCP/IP hosts to use. This control supports the TCP server side paradigm only, you must use the NetClient control to initiate a client connection to a remote host. You must also use the NetClient control to actually talk to any remote host that the **NetServer** control accept()s. A UDP server cannot yet be written with the current version of this control.

Distribution Note When you create and distribute applications written with the NetServer control, you should copy the file RELEASE.VBX as WSANET.VBX into your user's \SYSTEM subdirectory or otherwise in the system path. The WSANET.VBX file distributed with the project is a debugging version (and thus requires DBWIN or OX.SYS in order to work correctly).

Properties

These are the properties for the **NetServer** control. Properties that apply *only* to this control, or that require special consideration when using it, are marked with an asterisk (*).

*About	hWnd	*LocalService	Top
CtlName(Name)	Index	Parent	*Version
*Debug	Left	*QueueSize	
*ErrorMessage	*Listen	*Socket	
*ErrorNumber	*LocalPort	Tag	

Listen is the default property

Methods

This current version of the NetServer control supports no custom Methods.

Events

The NetServer control supports 2 events:

OnAccept() OnError()

Errors

This current version of the NetServer control returns these errors:

VBERR_INVALIDPROPERTYVALUE	380
VBERR_READONLY	383
VBERR_WRITEONLY	394
WSAERR_AlreadyConn	20001
WSAERR_NotConn	20005

You can include the "constants" file WSANET.TXT to get these `Global Const`'s.

Examples

The NetServer control currently has 2 sample applications:

WFingerd	A simple Finger Daemon/Server that can accept/handle 1 connection at a time.
VB-SMTPD	A complex SMTP Daemon/Server that can accept/handle multiple connections at a time, each connection contains it's own "state".

Debug Property, NetServer Control

Description

Sets or returns the current state of the SO_DEBUG socket option on the main listen()ing socket.

Visual Basic

[*form .*] *NetServer*.**Debug** [= *bool%*]
[*bool%*] = [*form .*] *NetServer*.**Debug**

Visual C++

pNetServer->**GetNumProperty("Debug")**
pNetServer->**SetNumProperty("Debug",***bool%***)**

Remarks

This property reflects the current socket level debugging state of the main listening socket.

When setting this property, if Listen is False, WSAERROR_NotConn (20005) is set by the control, and no action is taken. If Listen is True, setsockopt() is called, or the ErrorNumber property is set to the appropriate WinSock error.

When reading this property, if Listen is False, False is ALWAYS returned. If Listen is True, getsockopt() is called, or the ErrorNumber property is set to the appropriate WinSock error.

The debugging information supplied by most WinSock vendors is in the form of OutputDebugString() calls. In order to view this debugging information, you will need the DBWIN application from the 3.1 SDK, or an equivalent application that hooks the debugging output. Most commercial packages do NOT include any debugging information, or support the SO_DEBUG socket option.

WinSock calls

getsockopt(), setsockopt()

Data Type

Integer (Boolean)

Scope

VB1.0+/VC++

Listen Property, NetServer Control

Description

Reports or sets the current connection status of the associated socket.

Visual Basic

[*form .*] *NetServer*.**Listen**[= *state%*]
[*state%*] = [*form .*] *NetServer*.**Listen**

Visual C++

pNetServer->**GetNumProperty("Listen")**
pNetServer->**SetNumProperty("Listen",** *state***)**

Remarks

Setting this property to True (-1 or <> 0 and <> 2) will cause the control to start listening to the specified LocalPort. The port is allocated with SO_REUSEADDR, so be sure to remind the user that your daemon/server WILL take over the corresponding port. This "feature" was created just in case the application inadvertantly crashes without closing the Socket.

Setting this property to LISTEN_DONTREUSE (2) will cause the control to start listening to the specified LocalPort, but will NOT set the SO_REUSEADDR socket option. This is intended for programs that do not wish to take over the port of a service that is already handled by another application.

Setting this property to False (0) will cause the control to stop listening for any further incoming connections via the closesocket() call.

WinSock Calls

socket(), WSAAsyncSelect(), bind(), listen(), closesocket()

Data Type

Integer

Scope

VB1.0+/VC++

ErrorMessage Property, NetServer Control

See Also

Description

Returns the error message for the ErrorNumber property.

Visual Basic

[*message$*] = [*form .*] *NetServer*.**ErrorMessage**

Visual C++

pNetServer->**GetStrProperty("ErrorMessage")**

Remarks

This property corresponds to the ErrorNumber property. The VBX's resource string table is used to get the corresponding error message.

WinSock Calls

None.

Data Type

String (HSZ)

Scope

VB1.0+/VC++

See also

ErrorNumber
OnError()

ErrorNumber Property, NetServer Control

See Also

Description

Reports any error condition from the NetServer control.

Visual Basic

[*form .*] *NetServer*.**ErrorNumber** [= *status%*]
[*status%*] = [*form .*] *NetServer*.**ErrorNumber**

Visual C++

pNetServer->**GetNumProperty("ErrorNumber")**
pNetServer->**SetNumProperty("ErrorNumber",***status%***)**

Remarks

This property returns any error at the Windows Sockets level. You can set this property at any time in your application. The ErrorMessage property returns strings based upon the value stored in this property.

Please consult WinSock Errors for more information on WinSock specific errors.

WinSock Calls

WSAGetLastError() is called to get the error status whenever a WinSock function returns SOCKET_ERROR. Therefore, this property does not call any WinSock related functions; **ErrorNumbe**r is set as the error occurs.

Data Type

Integer

Scope

VB1.0+/VC++

See also

ErrorMessage
OnError()

LocalPort Property, NetServer Control

See Also

Description

The **LocalPort** property is used to set the local port number for the NetServer control to listen() to for an incoming TCP connection.

Visual Basic

[*form .*] *NetServer*.**LocalPort**[= *port%*]
[*port%*] = [*form .*] *NetServer*.**LocalPort**

Visual C++

pNetServer->**GetNumProperty("LocalPort")**
pNetServer->**SetNumProperty("LocalPort",** *port%***)**

Remarks

This property denotes the local port that is used to bind() a listen()ing NetServer Socket to. Any port number from 1 to 32767 can be used, altough the ports between 1 and 1024 are usually reserved for "restricted port" TCP services.

The idea of *restricted* ports stems from BSD Unix, where an application must run as root in order to bind() to a port below 1023. These ports are the well known Unix services such as FTP, SMTP, Finger, and so on.

Note: You MUST set this property before using the Listen property.

WinSock Calls

This property is used at listen() time. See the Listen property for more information.

Data Type

Integer

Scope

VB1.0+/VC++

See also

- LocalService
- Listen
- QueueSize

LocalService Property, NetServer Control

See Also

Description

This property allows an application to use the getservbyXXX() database services. By using this property, you allow the user to merely modify their TCP/IP's stack in order to use the port that your application uses for the local client port binding.

Visual Basic

[*form .*]*NetServer .***LocalService**[= *name$*]
[*name$*] = [*form .*] *NetServer.***LocalService**

Visual C++

pNetServer->**GetStrProperty("LocalService")**
pNetServer->**SetStrProperty("LocalService",** *name$***)**

Remarks

This property is used instead of the LocalPort property in order to set the *Service name* for the local port.

When set, this property calls getservbyname() and sets LocalPort to the corresponding integer port number.

When you get this properties' value it calls getservbyport() and returns the appropriate *Service name* for the LocalPort property.

When you either set or get this property, check it for an empty string (""). When the *Service* database lookup fails with a NULL pointer, this property will be set to "", and LocalPort will be set to 0.

The only real reason for using this property is to use reserved ports. You should use this call instead of the LocalPort property - not use a hard set value. This will allow your users to merely change their HOSTS file (or equivalent) to change the port that the daemon runs on. You may also decide to allow the user to set a desired port via your own application's setup dialog instead of their HOSTS file.

Some Vendors' WinSock.DLLs will appropriately block if the "Services" database is not held locally (i.e. Sun's NIS paradigm). Most Vendors implement their WinSock.DLLs in a non-blocking manner, but to do the opposite is quite within the WinSock v1.1 specification; thus, you ***may*** have a perceptible delay when this property is accessed.

WinSock Calls

getservbyname(), getservbyport()

Data Type

String (HSZ)

Scope

VB1.0+/VC++

See also

- LocalPort
- Listen
- QueueSize

QueueSize Property, NetServer Control

See Also

Description

The **QueueSize** property is used to set the number of allowed "queued" incoming connections. This is the exact value given to the listen() call.

Visual Basic

[*form .*] *NetServer*.**QueueSize**[= *size%*]
[*size%*] = [*form .*] *NetServer*.**QueueSize**

Visual C++

pNetServer->**GetNumProperty("QueueSize")**
pNetServer->**SetNumProperty("QueueSize",** *size%***)**

Remarks

You MUST set this property before you set the Listen property to True. If you set this property afterwards, it has no effect.

The default Windows Sockets v1.1 value for this property is 5.

This value currently has little effect other than to allow multiple clients to simulaneously connect to the listen() - after which one accept() at a time is handled.

Note: I can see no reason to change this value under the v1.1 spec, but I have included it *just in case*.

WinSock Calls

This property is used when listen() is called when the Listen property is set to True.

Data Type

Integer

Scope

VB1.0+/VC++

See also

- LocalPort
- LocalService
- Listen

Socket Property, NetServer Control

Description

Socket specifies the actual socket that is listening for incoming connections.

Visual Basic

[*form .*] *NetServer*.**Socket**=[*socket%*]
[*size%*] = [*form .*] *NetServer*.**Socket**

Visual C++

pNetServer->**GetNumProperty("Socket")**
pNetServer->**SetNumProperty("Socket",** *socket%***)**

Remarks

If Listen is False, this property has no meaning when read. The socket() call is not made until the Listen property is set.

If Listen is False, and you set this property to either 0 or SOCKET_ERROR (-1), the Listen property will be set to False - regardless.

There is no real reason to set the **Socket** property in the NetServer control, so it is currently NOT allowed (and will set the VB error VBERR_READONLY (383)).

Do *NOT* attempt a closesocket() on this handle. It will not currently have any adverse affects (other than sporadic errors), but it might in future versions. Do *NOT* set this **Socket** to blocking, and do *NOT* do a WSAAsyncSelect() on **Socket**, as it will surely break the current control.

For further reference, you may want to consult What is a Socket? and the WinSock.HLP file.

WinSock Calls

None.

Data Type

Integer

Scope

VB1.0+/VC++

Version Property, NetServer Control

Description

This property reports the current revision of the NetServer control. The current version of *THIS* WSANET.VBX distribution will return "v1.08alpha" as the version string.

Visual Basic

[*version$*] = [*form .*] *NetServer*.**Version**

Visual C++

pNetServer->**GetStrProperty("Version")**

Remarks

This property can *NOT* be set.

You can use this property at run time to warn the user that the application may, *or may not*, work with any newer version of the control. The NetServer control currently uses only 1 model - VB1.0 (although it does trick newer versions into thinking that it is VB2.0 in order to allow context sensitive help). Because of this single model approach, there should be no future incompatibilities with versions of this control.

This string is merely a way for programmers to deal with known incompatibilities of WSANET.VBX versions, and to report version information to the user. This is *NOT* the way for an application to realize the version of WinSock it is using - this version of WSANET does *NOT* support any version beside WinSock v1.1 (although future versions of certain Vendor's WINSOCK.DLLs may support the earlier v1.1 API).

WinSock Calls

None.

Data Type

String (Hsz)

Scope

VB1.0+/VC++

See Also
OnAccept
OnError

Using Events

OnAccept Event, NetServer Control

See Also

Description

Generated whenever a connection is accepted from a remote host.

Visual Basic

Sub *NetServer*_**OnAccept(**[*Index* **As Integer**], Socket **As Integer**, PeerAddr **As String**, RemotePort **As Integer)**

Visual C++

Function Signature:
void *CMyDialog::On Outline OnAccept***(UINT, int, CWnd*, LPVOID lpParams)**

Parameter Usage:
I do not know how to access multiple VBX event parameters with the MFC2.0 classes.

Remarks

The **OnAccept()** event is generated whenever a connection from a remote host is accept()ed.

After this event is fired, you shold hand the Socket event parameter to a NetClient control's Socket property in order to continue a connection. If, by checking the PeerAddr event parameter, you do not wish to talk to that host, you should set the Socket event parameter to -1 (SOCKET_ERROR) or 0 to close the accept()ed Socket.

The following example code shows a simple OnAccept() event handler:

```
Sub NetServer_OnAccept (Socket As Integer, PeerAddr As String, RemotePort As
                        Integer)
    On Error Resume Next
    If NetClient.Connect = True Then
        Socket = 0          ' Don't allow multiple fingers
        Exit Sub
    End If
    NetClient.Socket = Socket
End Sub
```

WinSock Events

This event is fired on response to an FD_ACCEPT message.

OnError Event, NetServer Control

See Also

Description

Generated whenever an error of some kind occurs in the NetServer control.

Visual Basic

Sub *NetServer*_**OnError(**[*Index* **As Integer**], ErrorNumber **As Integer)**

Visual C++

Function Signature:
void *CMyDialog::On Outline OnError***(UINT, int, CWnd*, LPVOID lpParams)**

Parameter Usage:
I do not know how to access multiple VBX event parameters with the MFC2.0 classes.

Remarks

The **ErrorNumber** value and ErrorMessage property can be used to identify the error that occurred.

WinSock Events

This event is fired whenever an error occurs. WSAGetLastError() is called by the NetServer control if the error was WinSock related before the **OnError()** event is fired.

WinSock Network Client Control

Properties Methods Events Errors Examples

Description

You can use the NetClient control to access Windows Sockets v1.1 services. This control is invisible at run time, much like the timer control provided with Visual Basic.

File name

WSANET.VBX

Object Type

NetClient

Remarks

The NetClient control uses functionality provided by a vendor specific version of WINSOCK.DLL that is provided with your TCP/IP protocol kernel. With this control, you can access any TCP service that is running on a TCP/IP host. This control supports the TCP client side paradigm only, you must use the NetServer control in order to write server/daemon applications. UDP and OOB communication are not yet supported in this version.

See About this project for more general information
See the Changes that have been made to this version.
See the public domain Copyright information for distribution availability.

Distribution Note When you create and distribute applications written with the NetClient control, you should copy the file RELEASE.VBX as WSANET.VBX into your user's \SYSTEM subdirectory or otherwise in the system path. The WSANET.VBX file distributed with the project is a debugging version.

Properties

These are the properties for the NetClient Control. Properties that apply *only* to this client control, or that require special consideration when using it, are marked with an asterisk (*).

*About	*HostAliasCount	Parent	*SendLine
*Block	*HostAliasList	*RecvBlock	*SendSize
*Connect	*HostName	*RecvCount	*SendThreshold
*Debug	Index	*RecvLine	*Socket
*ErrorMessage	Left	*RecvSize	*TimeOut
*ErrorNumber	*Line	*RecvThreshold	Tag
*Host	*LineDelimiter	*RemotePort	Top
*HostAddr	*LocalPort	*RemoteService	*Version
*HostAddressCount	*LocalService	*SendBlock	
*HostAddressList	Name	*SendCount	

Line is the default property.

Methods

This current version of the NetClient control supports no custom Methods.

Events

The NetClient control uses 6 custom Events in order to support Asyncronous Windows Sockets.

OnConnect	OnRecv	OnSend	OnClose	OnError	OnTimeOut

Errors

Windows Sockets errors are returned by triggering the OnError() event, and setting the ErrorNumber and ErrorMessage properties appropriately.

The NetClient control currently sets the following errors:

ERR_OUTOFMEMORY	7
VBERR_INVALIDPROPERTYVALUE	380
VBERR_BADVBINDEX	381
VBERR_READONLY	383
VBERR_WRITEONLY	394
WSAERR_BadHostAddr	20000
WSAERR_AlreadyConn	20001
WSAERR_NoTimers	20002
WSAERR_RecvBuffer	20003
WSAERR_SendOverFlow	20004
WSAERR_NotConn	20005

These errors are all contained within the "constants" file WSANET.TXT

You MUST use On Error handlers to trap these errors.

Coding Examples

WSANET.VBX is origionally packaged with 5 example programs that show how to use the NetClient control effectively. All examples have been tested in VB2.0 and VB3.0, and are shown here in text saved format. Only the VB2.0 examples are included in this HLP file for space considerations.

WFinger		A simple finger client. Shows how to carry on a simple conversation.
WiNslookup	-	A NSLookup like application. Shows how to use DNS and I.P lookup properties.
WSMTPC	-	A simple SMTP client. Shows how to make a state-based client application.
WinWhois	-	A "NicName" or WHOIS client. Shows how to use the INI control, and the NetClient control in a real application.
WFingerD	-	A simple finger server. Shows how to use the NetServer control with 1 connection at a time service.
VB-SMTPD	-	A complex SMTP daemon. Shows how to create an unlimited connection state server with the NetServer control.

In order to use the above examples, you may locate the relevant materials and use WinHelp's Edit/Copy menu to copy the selected source directly into your code window.

About Property, NetClient & NetServer Controls

Description

Shows the Author and version of your current WSANET.VBX during design time.

Visual Basic

Design time only

Visual C++

Design time only.

Remarks

This property merely shows the current version "stepping" of WSANET.VBX and the author's name in a dialog box form. This should be of little importance to the application developer. For more project information, read About this project and Thanks

WinSock Calls

None.

Data Type

None. (Only a VBM_GETPROPHSZ response)

Scope

VB1.0+/VC++

Block Property, NetClient Control

See Also

Description

Sends or receives a block of raw binary data from/to a connected Socket.

Visual Basic

[*form .*]*NetClient* .**Block**[= *setting$*]
[*setting$*] = [*form .*] *NetClient*.**Block**

Visual C++

Visual C++ cannot Access Visual Basic strings. (HLSTR is implemented as a property only in VB2.00+) Consult the NetClientSendBlock() and NetClientRecvBlock() DLL function exports for more information.

Remarks

This property is nothing more than BOTH the SendBlock and RecvBlock properties combined into a single property. Setting this property invokes SendBlock, and getting this property invokes RecvBlock.

WinSock Calls

See the RecvBlock and SendBlock properties.

Data Type

String (HLSTR)

Scope

VB2.0+

See also

RecvBlock
SendBlock

Connect Property, NetClient Control

See Also

Description

Reports or sets the current connection status of the associated socket.

Visual Basic

[*form .*] *NetClient*.**Connect**[= *state%*]
[*state%*] = [*form .*] *NetClient*.**Connect**

Visual C++

pNetClient->**GetNumProperty("Connect")**
pNetClient->**SetNumProperty("Connect",** *state***)**

Remarks

Setting this property to True (<>0, or -1 in VB) will cause the control to try and connect to a remote host. Set this property to True only when you have set the RemotePort and HostName/HostAddr pair to the desired TCP port and machine name for the remote host.

If you set LocalPort to non-zero, a bind() will executed to bind the local end of the port to the desired port. If LocalPort is zero, no binding is performed, and the WinSock connect() call will bind() the local end to a benign port.

Setting this property to False (=0) will cause the control to terminate any ongoing connections with the remote host via the closesocket() call.

WinSock Calls

socket(), WSAAsyncSelect(), bind(), connect(), closesocket()

Data Type

Integer

Scope

VB1.0+/VC++

See also

- HostName
- HostAddr
- LocalPort
- RemotePort

Debug Property, NetClient Control

Description

Sets or returns the current state of the SO_DEBUG socket option.

Visual Basic

[*form .*] *NetClient*.**Debug** [= *bool%*]
[*bool%*] = [*form .*] *NetClient*.**Debug**

Visual C++

pNetClient->**GetNumProperty("Debug")**
pNetClient->**SetNumProperty("Debug",***bool%***)**

Remarks

This property reflects the current socket level debugging state of a connected socket.

When setting this property, if Connect is False, WSAERR_NotConn (error 20000) is set by the control, and no action is taken. If Connect is True, setsockopt() is called, or the ErrorNumber property is set to the appropriate WinSock error.

When reading this property, if Connect is False, False is ALWAYS returned. If Connect is True, getsockopt() is called, or the ErrorNumber property is set to the appropriate WinSock error.

The debugging information supplied by most WinSock vendors is in the form of OutputDebugString() calls. In order to view this debugging information, you will need the DBWIN application from the 3.1 SDK, or an equivalent application that hooks the debugging output.

WinSock calls

getsockopt(), setsockopt()

Data Type

Integer (Boolean)

Scope

VB1.0+/VC++

ErrorMessage Property, NetClient Control

See Also

Description

Returns the error message for the ErrorNumber property.

Visual Basic

[*message$*] = [*form .*] *NetClient*.**ErrorMessage**

Visual C++

pNetClient->**GetStrProperty("ErrorMessage")**

Remarks

This property corresponds to the ErrorNumber property. The VBX's resource string table is used to get the corresponding error message.

WinSock Calls

None.

Data Type

String (HSZ)

Scope

VB1.0+/VC++

See also

ErrorNumber
OnError()

ErrorNumber Property, NetClient Control

See Also

Description

Reports any error condition from the NetClient control.

Visual Basic

[*form .*] *NetClient*.**ErrorNumber** [= *status%*]
[*status%*] = [*form .*] *NetClient*.**ErrorNumber**

Visual C++

pNetClient->**GetNumProperty("ErrorNumber")**
pNetClient->**SetNumProperty("ErrorNumber",***status%***)**

Remarks

This property returns any error at the Windows Sockets level, or other miscellaneous errors caused by control usage error. You can set this property at any time in your application. The ErrorMessage property returns strings based upon the value stored in this property.

Please consult WinSock Errors for more information on WinSock specific errors.

WinSock Calls

WSAGetLastError() is called to get the error status whenever a WinSock function returns SOCKET_ERROR. Therefore, this property does not call any WinSock related functions; **ErrorNumbe**r is set as the error occurs.

Data Type

Integer

Scope

VB1.0+/VC++

See also

ErrorMessage
OnError()

Host Property, NetClient Control

See Also

Description

This property is used to resolve an I.P. number or DNS name, and set the remote host's address for socket communication.

Visual Basic

[*form .*] *NetClient*.**Host**[= *host$*]
[*host$*] = [*form .*] *NetClient*.**Host**

Visual C++

pNetClient->**GetStrProperty("Host")**
pNetClient->**SetStrProperty("Host",** hostaddr**)**

Remarks

When set, first a HostName lookup, and then a HostAddr lookup is attempted on the host name.

This property returns the same thing as HostName (it uses the same property handling code).

Please see the HostName and HostAddr properties for the full description.

This property is merely a "convenience" property for applications to avoid checking both of the above properties during a user input.

Currently, setting this property invokes the corresponding *blocking* call of your WinSock. This is not the best way to implement this functionality. A better way would be to add a OnGetHost() event and a HostAsync property. If this were used, WSAAsyncGetHostByAddr() could be used, and your application would not block. While your application *blocks*, your application will *stick* until it completes.

WinSock Calls

inet_ntoa(), inet_addr(), gethostbyaddr(), gethostbyname()

Data Type

String (HSZ)

Scope

VB1.0+/VC++

See also

- HostAddr
- HostName
- HostAddressCount
- HostAddressList
- HostAliasCount
- HostAlistList

HostAddr Property, NetClient Control

See Also

Description

This property is used to resolve an I.P. number and set the remote host's address for socket communication.

Visual Basic

[*form .*] *NetClient*.**HostAddr**[= *hostaddr$*]
[*hostaddr$*] = [*form .*] *NetClient*.**HostAddr**

Visual C++

pNetClient->**GetStrProperty("HostAddr")**
pNetClient->**SetStrProperty("HostAddr",** hostaddr**)**

Remarks

At run time, this property initially holds the I.P. address of your local host (or "" if your WinSock is substandard).

Setting this property to a host number string invokes the name resolver. Your program will temporarily block (for often no more than a fraction of a second) while a DNS lookup is made through your local name server or local *hosts* file on your PC. If an error occurs, this property will be set to a empty string ("") , the HostName property will be set to an empty string ("") and the OnError() event will be called with the appropriate WinSock error. If no error occurs, this property will be set to the I.P. address of the host, and the HostName property will be set to the correct FQDN of the host.

Reading this property will be the result of any previous *set* operation, or your local PC's I.P. address if it is the first time you access it.

If you use this property on a Connected session, you will loose all reference to the machine that the control *was* talking to. The socket communications will proceed, but it becomes your application's responsibility to know who it is talking to.

You only need to set **HostAddr** OR HostName; setting both is unnecessarily redundant. After setting one of these properties, the other is set appropriately.

If the requested Host is in your WinSock.DLL's database (most likely from a file named "hosts" in its path), no blocking *should* be involved - but this seems to be vendor specific. If your vendor's WinSock.DLL does not block as it reads its database, there should be no actual delay as normal DNS lookups would incur.

Currently, setting this property invokes the corresponding *blocking* call of your WinSock. This is not the best way to implement this functionality. A better way would be to add a OnGetHost() event and a HostAsync property. If this were used, WSAAsyncGetHostByAddr() could be used, and your application would not block. While your application *blocks*, your application will *stick* until it completes.

WinSock Calls

inet_ntoa(), inet_addr(), gethostbyaddr()

Data Type

String (HSZ)

Scope

VB1.0+/VC++

See also

HostName
HostAddressCount
HostAddressList
HostAliasCount
HostAlistList

HostAddressCount Property, NetClient Control

See Also

Description

This property holds the current number of elements in the HostAddressList property array. This number signifies the number of I.P. Addresses for a DNS host.

Visual Basic

[*count%*] = [*form .*] *NetClient*.**HostAddressCount**

Visual C++

pNetClient->**GetNumProperty("HostAddressCount")**

Remarks

When you set HostAddr or HostName, this property is set automagically.

Setting **HostAddressCount** will return VBERR_WRITEONLY (383).

Reading **HostAddressCount** will tell you the number of I.P. strings contained in the HostAddressList() property array. Do NOT attempt to read past this limit, or you will receive VBERR_BADVBINDEX (381).

WinSock Calls

None. This property calls no actual WinSock calls. The number returned is from the number of h_address_list[] entries in the HOSTENT structure returned by gethostbyname()/gethostbyaddr().

Data Type

Integer

Scope

VB1.0+/VC++

See also

- HostAddr
- HostName
- HostAddressList
- HostAliasCount
- HostAliasList

HostAddressList Property, NetClient Control

See Also

Description

This property array holds the I.P. strings returned via gethostbyname()/gethostbyaddr().

Visual Basic

[*hostaddr$*] = [*form .*] *NetClient*.**HostAddressList(***index***)**

Visual C++

I do not know how VC++ uses property arrays.

Remarks

When you set HostAddr or HostName, this property is set automagically.

This property array holds the h_address_list[] array as returned through the HOSTENT structure by gethostbyname()/gethostbyaddr().

The number of elements of this array is available through the HostAddressCount property. Do NOT attempt to access the **HostAddressList()** array past this limit, or you will receive VBERR_BADVBINDEX (381).

By using the *index* to step through the array, you can find all I.P. addresses for a remote host. This helps multi-homed clients and applications that need to know multiple "routes" to a remote host.

HostAddressList(0) will ALWAYS hold the same value as HostAddr, so HostAddressCount will always be 1 if HostAddr holds a valid I.P string.

WinSock Calls

None. This property calls no actual WinSock calls. The number returned is from the actual h_address_list[] entries in the HOSTENT structure returned by gethostbyname()/gethostbyaddr().

Data Type

Array of **String**s

Scope

VB1.0+, VC++ possibly

See also

- HostAddr
- HostName
- HostAddressCount
- HostAliasCount
- HostAliasList

HostAliasCount Property, NetClient Control

See Also

Description

This property holds the current number of elements in the HostAliasList property array. This number signifies the number of I.P. Addresses for a DNS host.

Visual Basic

[*count%*] = [*form .*] *NetClient*.**HostAliasCount**

Visual C++

pNetClient->**GetNumProperty("HostAliasCount")**

Remarks

Setting **HostAliasCount** will return VBERR_WRITEONLY (383).

Reading **HostAliasCount** will tell you the number of DNS host strings contained in the HostAliasList() property array. Do NOT attempt to read past this limit, or you will receive VBERR_BADVBINDEX (381).

WinSock Calls

None. This property calls no actual WinSock calls. The number returned is from the number of h_aliases[] entries in the HOSTENT structure returned by gethostbyname()/gethostbyaddr(). When you set HostAddr or HostName, this property is set automagically.

Data Type

Integer

Scope

VB1.0+/VC++

See also

- HostAddr
- HostName
- HostAliasList
- HostAddressCount
- HostAddressList

HostAliasList Property, NetClient Control

See Also

Description

This property array holds the DNS strings returned via gethostbyname()/gethostbyaddr().

Visual Basic

[*hostaddr$*] = [*form .*] *NetClient*.**HostAliasList(***index***)**

Visual C++

I do not know how VC++ uses property arrays.

Remarks

When you set HostAddr or HostName, this property is set automagically.

This property array holds the h_aliases[] array as returned through the HOSTENT structure by gethostbyname()/gethostbyaddr().

The number of elements of this array is available through the HostAliasCount property. Do NOT attempt to access the **HostAliasList()** array past this limit, or you will receive VBERR_BADVBINDEX (381).

By using the *index* to step through the array, you can find all DNS addresses for a remote host. This helps multi-homed clients and applications that need to know multiple "routes" to a remote host.

WinSock Calls

None. This property calls no actual WinSock calls. The number returned is from the actual h_aliases[] entries in the HOSTENT structure returned by gethostbyname()/gethostbyaddr().

Data Type

Array of **String**s

Scope

VB1.0+, VC++ possibly

See also

- HostAddr
- HostName
- HostAliasCount
- HostAddressCount
- HostAddressList

HostName Property, NetClient Control

See Also

Description

This property is used to resolve a DNS name and set the remote host's FQDN for socket communication.

Visual Basic

[*form .*] *NetClient*.**HostName**[= *hostname$*]
[*fullhostname$*] = [*form .*] *NetClient*.**HostName**

Visual C++

pNetClient->**GetStrProperty("HostName")**
pNetClient->**SetStrProperty("HostName",** hostname**)**

Remarks

At run time, this property initially holds the FQDN of your local host (or "" if your WinSock is substandard).

Setting this property to a host name invokes the name resolver. Your program will temporarily block (for often no more than a fraction of a second) while a DNS lookup is made through your local name server or local *hosts* file on your PC. If an error occurs, this property will be set to a empty string ("") , the HostAddr property will be set to an empty string ("") and the OnError() event will be called with the appropriate WinSock error. If no error occurs, this property will be set to the FQDN of the host, and the HostAddr property will be set to the correct I.P. string for the host.

Reading this property will be the result of any previous *set* operation, or your local PC's FQDN if it is the first time you access it.

You only need to set HostAddr OR **HostName**; setting both is unnecessarily redundant. After setting one of these properties, the other is set appropriately.

If the requested Host is in your WinSock.DLL's database (most likely from a HOSTS file), no blocking *should* be involved - but this seems to be vendor specific. If your vendor's WinSock.DLL does not block as it reads its database, there should be no actual delay as normal DNS lookups would incur.

Currently, setting this property invokes the corresponding *blocking* call of your WinSock. This is not the best way to implement this functionality. A better way would be to add a OnGetHost() event and a HostAsync property. If this were used, WSAAsyncGetHostByName() could be used, and your application would not block. While your application *blocks*, your application will *stick* until it completes.

WinSock Calls

inet_ntoa(), gethostbyname()

Data Type

String (HSZ)

Scope

VB1.0+/VC++

See also

HostAddr

Line Property, NetClient Control

See Also

Description

Sends or receives a **Line** of data from/to a connected Socket.

Visual Basic

[*form .*]*NetClient* .**Line**[= *lineoutput$*]
[*lineinput$*] = [*form .*] *NetClient*.**Line**

Visual C++

pNetClient->**GetStrProperty("Line")**
pNetClient->**SetStrProperty("Line",** *lineoutput$***)**

Remarks

This property is nothing more than BOTH the SendLine and RecvLine properties combined into a single property. Setting this property invokes SendLine, and getting this property invokes RecvLine.

WinSock Calls

See the RecvLine, SendLine, and LineDelimiter properties.

Data Type

String (HSZ)

Scope

VB1.0+/VC++

See also

RecvLine
LineDelimiter
SendLine

LineDelimiter Property, NetClient Control

See Also

Description

This property is tied to the functionality of the RecvLine property and determines the end-of-line marker for incoming data.

Visual Basic

[*form .*]*NetClient* .**LineDelimiter**[= *eol$*]
[*eol$*] = [*form .*] *NetClient*.**LineDelimiter**

Visual C++

pNetClient->**GetStrProperty("LineDelimiter")**
pNetClient->**SetStrProperty("LineDelimiter",** *eol$***)**

Remarks

The **LineDelimiter** property is used to specify the end-of-line character sequence for incoming data accessed through the RecvLine property. When set to a non-empty sequence of characters (except Chr$(0)), this property dictates what can be seen in the RecvLine property. When an application accesses the RecvLine property, the incoming receive buffer is scanned for the character sequence in the **LineDelimiter** property and any if the sequence is found, the stream preceeding the end-of-line sequence is returned. The end-of-line sequence, when and if found, *DOES NOT* appear at the end of any data received from the RecvLine property.

When **LineDelimiter** is set to an empty string (""), the RecvLine property will return the entire incoming buffer.

An example use of the **LineDelimiter** property might be:

```
NetClient(0).LineDelimiter = Chr$(13) + Chr$(10)
```

This would set up any reads from RecvLine to return incoming lines that are terminated by a Carriage Return and Line Feed sequence, respectively. This would be the appropriate setting for talking to another DOS host, or protocol such as SMTP which specifies both characters as a end-of-line marker. On most Unix style servers, the default end-of-line sequence is Chr$(10) ("\n").

Translated sequences (for C programmer's convenience):

String	VB Equivalent	ASCII meaning	ASCII "Keystroke"
"\n"	Chr$(10)	Linefeed	Control J
"\r"	Chr$(13)	Carriage Return	Control M
"\t"	Chr$(8)	Horizontal Tab	Control H

Please consult the RecvLine and Line properties for more information.

NOTE: Due to the nature of this property, it may NOT contain an embedded NULL character (Chr$(0)). If your data depends on binary data, you will have to use the Block/RecvBlock/SendBlock properties.

Data Type

String (HSZ)

Scope

VB1.0+/VC++

See also

RecvLine
Line

LocalPort Property, NetClient Control

See Also Thanks

Description

The **LocalPort** property is used to set the local port number for a TCP connection.

Visual Basic

[*form .*] *NetClient*.**LocalPort**[= *port%*]
[*port%*] = [*form .*] *NetClient*.**LocalPort**

Visual C++

pNetClient->**GetNumProperty("LocalPort")**
pNetClient->**SetNumProperty("LocalPort",** *port%***)**

Remarks

This property denotes the local port for the TCP connection to a remote port. Some TCP daemons will only accept connections from programs that are locally bound to a port below 1023. In order to make your application compliant to such restrictions, you should loop backwards from 1023 to 512 in order to allocate a unique *restricted* port.

The idea of *restricted* ports stems from BSD Unix, where an application must run as root in order to bind() to a port below 1023. Setting this property to 0 will allow the WinSock.DLL to bind the local port to a unique port between 1023 and 5000. This functionality is accomplished by not calling bind() at connect() time.

Here is a portion of code from a fictitious RSH server that allocates a *restricted* port:

```
On Error Resume Next

NetClient.RemotePort = 512
iPort = 1023

Do
    NetClient.LocalPort = 1023
    NetClient.ErrorNumber = 0   ' Clear ErrorNumber
    NetClient.Connect = True
    If NetClient.ErrorNumber = 0 Then Exit Sub
    NetClient.ErrorNumber = 0
    iPort = iPort - 1
    If iPort = 512 Then
         TextTTY = "Error! Ran out of reserved ports to bind to!"
         Exit Sub
    End If
    DoEvents
Loop
```

Note: If you are not using reserved ports for your client, you should set this property to 0 to allow dynamic port allocation.

WinSock Calls

This property is used at connect() time. See the Connect property for more information.

Data Type

Integer

Scope

VB1.0+/VC++

See also

- LocalService
- RemotePort
- HostName
- HostAddr
- Connect

LocalService Property, NetClient Control

See Also

Description

This property allows an application to use the getservbyXXX() database services. By using this property, you allow the user to merely modify their TCP/IP's stack in order to use the port that your application uses for the local client port binding.

Visual Basic

[*form .*]*NetClient* .**LocalService**[= *name$*]
[*name$*] = [*form .*] *NetClient*.**LocalService**

Visual C++

pNetClient->**GetStrProperty("LocalService")**
pNetClient->**SetStrProperty("LocalService",** *name$***)**

Remarks

This property is not as useful as the RemoteService property in that you will probably never need to allow the user to set the local client port binding. If you need to use reserved ports, which is the only real reason for using this or the LocalPort property, you will probably need to loop within a range of reserved ports - not use a hard set value from your PC's *Service* database.

This property is used instead of the LocalPort property in order to set the *Service name* for the local port.

When set, this property calls getservbyname() and sets LocalPort to the corresponding integer port number.

When you get this properties' value it calls getservbyport() and returns the appropriate *Service name* for the LocalPort property.

When you either set or get this property, check it for an empty string (""). When the *Service* database lookup fails with a NULL pointer, this property will be set to "", and LocalPort will be set to 0.

Some Vendors' WinSock.DLLs will appropriately block if the "Services" database is not held locally. Most Vendors implement their WinSock.DLLs in a non-blocking manner, but to do the opposite is quite within the WinSock v1.1 specification; thus, you may have a perceptible delay when this property is accessed.

WinSock Calls

getservbyname(), getservbyport()

Data Type

String (HSZ)

Scope

VB1.0+/VC++

See also

LocalPort
RemotePort

RecvBlock Property, NetClient Control

See Also

Description

RecvBlock allows an application to talk in pure binary to a remote host.

Visual Basic

[*message$*] = [*form .*] *NetClient*.**RecvBlock**

Visual C++

Visual C++ cannot Access Visual Basic strings. (HLSTR is implemented as a property only in VB2.00+) Consult the NetClientSendBlock() and NetClientRecvBlock() DLL function exports for more information.

Remarks

RecvBlock uses the newer HLSTR property type (introduced in VB2.00+). Unlike the RecvLine property with LineDelimiter set to "", embedded Null characters (Chr$(0)) is permitted.

RecvBlock will return whatever is in the receive buffer as a VB string.

Visual C++ does NOT implement the HLSTR type as a property, so it must use the functions NetClientSendBlock() and NetClientRecvBlock() that are exported from the VBX itself (as a VBX is nothing more than a DLL).

WinSock Calls

None. This property calls no actual WinSock calls. The recv() call is used during an FD_READ event generated to the control window via WSAAsyncSelect(), and the data is buffered to be returned through this property on demand (usually in response to a OnRecv() event).

Data Type

String (HLSTR)

Scope

VB2.0+

See also

Block
SendBlock

RecvCount Property, NetClient Control

See Also

Description

This property holds the current number of bytes awaiting processing in the Receive FIFO buffer.

Visual Basic

[*count%*] = [*form .*] *NetClient*.**RecvCount**

Visual C++

pNetClient->**GetNumProperty("RecvCount")**

Remarks

Setting **RecvCount** has no effect.

This property changes as bytes are received on a connected socket. If the number of bytes received exceed the limit in RecvThreshold, the OnRecv() event is fired. If RecvThreshold is 0, OnRecv() will be fired whenever more incoming bytes are received.

WinSock Calls

None. This property calls no actual WinSock calls. The recv() call is used during an FD_READ event generated to the control window via WSAAsyncSelect(), and the data is buffered to be returned on demand (usually in response to a OnRecv() event). This property merely reports the current number of characters in the receive buffer.

Data Type

Integer

Scope

VB1.0+/VC++

See also

SendCount
RecvSize
SendSize
RecvThreshold
SendThreshold

RecvLine Property, NetClient Control

See Also

Description

Receives a line of data from a connected Socket.

Visual Basic

[*lineinput$*] = [*form .*] *NetClient*.**RecvLine**

Visual C++

pNetClient->**GetStrProperty("RecvLine")**

Remarks

This property receives a single line of data from the receive buffer. The end-of-line sequence should first be specified through the LineDelimiter property in order for this property to only return a line at a time. If the LineDelimiter property is set to an empty string (""), then this property will return all non-null characters from the buffer. If you need to receive binary data, you should use the Block/RecvBlock/SendBlock properties or NetClientSendData()/NetClientRecvData().

If you have set the LineDelimiter property to a non empty string, you should set up a loop in your program to handle any other lines that might have been received as well. Your application should make sure that **RecvLine** returns an empty string before proceeding, unless it is prepared to handle the received data out of the OnRecv() event.

WinSock Calls

None. This property calls no actual WinSock calls. The recv() call is used during an FD_READ event generated to the control window via WSAAsyncSelect(), and the data is buffered to be returned through this property on demand (usually in response to a OnRecv() event).

Data Type

String (HSZ)

Scope

VB1.0+/VC++

See also

RecvThreshold
SendLine
Line

RecvSize Property, NetClient Control

See Also

Description

The **RecvSize** property is used to set size of the receive FIFO buffer.

Visual Basic

[*form .*] *NetClient*.**RecvSize**[= *size%*]
[*size%*] = [*form .*] *NetClient*.**RecvSize**

Visual C++

pNetClient->**GetNumProperty("RecvSize")**
pNetClient->**SetNumProperty("RecvSize",** *size%***)**

Remarks

By default, this property holds the value returned by a getsockopt() call for SO_RCVBUF immediately before an OnConnect() event, or a saved value from design time, which ever is greater.

Setting this property will force the NetClient control to TRY and resize the receive FIFO buffer size to size specified.

Getting this property will return the CURRENT size of the receive FIFO buffer.

WinSock Calls

If there is data in the receive buffer, the OnRecv() event will be fired in order to let the application attempt to parse the data. If the receive buffer is then empty, or was empty initially empty, the Global buffer is reallocated.

Data Type

Integer

Scope

VB1.0+/VC++

See also

SendSize
SendCount
RecvCount
RecvThreshold
SendThreshold

RecvThreshold Property, NetClient Control

See Also

Description

This property specifies how many characters are allowed to be queued in the receive buffer before the OnRecv() event is fired.

Visual Basic

[*form .*] *NetClient*.**RecvThreshold**[= *size%*]
[*size%*] = [*form .*] *NetClient*.**RecvThreshold**

Visual C++

pNetClient->**GetNumProperty("RecvThreshold")**
pNetClient->**SetNumProperty("RecvThreshold",** *size%***)**

Remarks

By default, this property is set to 0. This means that every FD_READ message will result in a OnRecv() event if characters are received.

This property can be set to any value in order to allow your application to cut down on the number of OnRecv() events that it received.

This property is of enormous use to those programs that send and expect large *chunks* of data. If your minimum transmission size is 64k, you wouldn't need to handle mere 512 byte packets at a time. By setting this property to the smallest aggregate transmission size, you allow the network control to receive a number of incoming stream fragments without waiting for an OnRecv() event to finish.

WinSock Calls

None.

Data Type

Integer

Scope

VB1.0+/VC++

See also

SendThreshold
SendSize
RecvSize
SendCount
RecvCount

RemotePort Property, NetClient Control

See Also

Description

The **RemotePort** property is used to set the remote port number for a TCP connection.

Visual Basic

[*form .*] *NetClient*.**RemotePort**[= *port%*]
[*port%*] = [*form .*] *NetClient*.**RemotePort**

Visual C++

pNetClient->**GetNumProperty("RemotePort")**
pNetClient->**SetNumProperty("RemotePort",** *port%***)**

Remarks

This property denotes the remote port for the TCP connection to a remote host. The **RemotePort** is specified at Connect time, after which it may NOT be changed during a connection.

This property MUST be set either directly, or through the RemoteService property, before a remote connection can occur.

You may also want to specify the LocalPort property if your application is acting as a restricted port client.

WinSock Calls

This property is used at connect() time. See the Connect property for more information.

Data Type

Integer

Scope

VB1.0+/VC++

See also

LocalPort
RemoteService

RemoteService Property, NetClient Control

See Also

Description

This property allows an application to use the getservbyXXX() database services. By using this property, you allow the user to merely modify their TCP/IP's stack in order to use the port that your application uses for the RemotePort binding.

Visual Basic

[*form .*]*NetClient* .**RemoteService**[= *name$*]
[*name$*] = [*form .*] *NetClient*.**RemoteService**

Visual C++

pNetClient->**GetStrProperty("RemoteService")**
pNetClient->**SetStrProperty("RemoteService",** *name$***)**

Remarks

This property is used instead of the RemotePort property in order to set the *Service name* for the remote port.

When set, this property calls getservbyname() and sets RemotePort to the corresponding integer port number.

When you get this properties' value it calls getservbyport() and returns the appropriate *Service name* for the RemotePort property.

When you either set or get this property, check it for an empty string (""). When the *Service* database lookup fails with a NULL pointer, this property will be set to "", and RemotePort will be set to 0.

Some Vendors' WinSock.DLLs will appropriately block if the "Services" database is not held locally. Most Vendors implement their WinSock.DLLs in a non-blocking manner, but to do the opposite is quite within the WinSock v1.1 specification; thus, you may have a perceptible delay when this property is accessed.

WinSock Calls

getservbyname(), getservbyport()

Data Type

String (HSZ)

Scope

VB1.0+/VC++

See also

LocalPort
RemotePort

SendBlock Property, NetClient Control

See Also

Description

SendBlock allows an application to talk in pure binary to a remote host.

Visual Basic

[*form .*] *NetClient*.**SendBlock** =[*block$*]

Visual C++

Visual C++ cannot access Visual Basic strings. (HLSTR is implemented as a property only in VB2.00+) Consult the NetClientSendBlock() and NetClientRecvBlock() DLL function exports for more information.

Remarks

SendBlock uses the newer HLSTR property type (introduced in VB2.00+). Unlike the SendLine property with LineDelimiter set to "", embedded Null characters (Chr$(0)) is permitted.

SendBlock will put whatever is handed to it as a VB string into in the send buffer.

Visual C++ does *NOT* implement the HLSTR type as a property, so it must use the functions NetClientSendBlock() and NetClientRecvBlock() that are exported from the VBX itself (as a VBX is nothing more than a DLL).

WinSock Calls

This property calls send() until either:

1) All of the handed data is accepted by the WinSock.DLL

2) The send() call returns WSAEWOULDBLOCK

In the second case, a flag is set so that no more send attempts are made until an FD_WRITE message is posted to the control's window by the WinSock.DLL. When this message is posted, the flag is cleared, and an attempt at sending unsent data is attempted.

Data Type

String (HLSTR)

Scope

VB2.0+

See also

RecvBlock
Block

SendCount Property, NetClient Control

See Also

Description

This property holds the current number of bytes waiting to be sent in the Send FIFO buffer.

Visual Basic

[*count%*] = [*form .*] *NetClient*.**SendCount**

Visual C++

pNetClient->**GetNumProperty("SendCount")**

Remarks

Setting **SendCount** has no effect.

This property changes as bytes are sent to a connected socket. If the number of bytes sent reach the limit in SendThreshold, the OnSend() event is fired.

WinSock Calls

None. This property calls no actual WinSock calls. This property merely reports the current number of characters in the send buffer.

Data Type

Integer

Scope

VB1.0+/VC++

See also

- RecvCount
- RecvSize
- SendSize
- RecvThreshold
- SendThreshold

SendLine Property, NetClient Control

See Also

Description

SendLine allows an application to talk in ASCII text to a remote host.

Visual Basic

[*form .*] *NetClient*.**SendLine** =[*lineoutput$*]

Visual C++

pNetClient->**SetStrProperty("SendLine",** *lineoutput$***)**

Remarks

SendLine uses the older HSZ property type. Unlike the SendBlock property, embedded Null characters (Chr$(0)) are *NOT* permitted.

SendLine will put whatever is handed to it as a VB string into in the send buffer.

SendLine does NOT use the LineDelimiter property; all strings sent through **SendLine** are sent as specified. You must include end-of-line terminators in the string you are sending.

You may send multiple lines at a time, but do not exceed the value of SendSize minus SendCount 'lest you chance a GPF.

WinSock Calls

This property calls send() until either:

1) All of the handed data is accepted by the WinSock.DLL

2) The send() call returns WSAEWOULDBLOCK

In the second case, a flag is set so that no more send attempts are made until an FD_WRITE message is posted to the control's window by the WinSock.DLL. When this message is posted, the flag is cleared, and an attempt at sending unsent data is again attempted.

Data Type

String (HSZ)

Scope

VB2.0+

See also

RecvBlock
Block

SendSize Property, NetClient Control

See Also

Description

The **SendSize** property is used to set the size of the receive FIFO buffer.

Visual Basic

[*form .*] *NetClient*.**SendSize**[= *size%*]
[*size%*] = [*form .*] *NetClient*.**SendSize**

Visual C++

pNetClient->**GetNumProperty("SendSize")**
pNetClient->**SetNumProperty("SendSize",** *size%***)**

Remarks

By default, this property holds the value returned by a getsockopt() call for SO_SNDBUF immediately before an OnConnect() event, or a saved value from design time, which ever is greater.

Setting this property will force the NetClient control to TRY and resize the send FIFO buffer size to the size specified.

Getting this property will return the CURRENT size of the send FIFO buffer.

WinSock Calls

If there is data in the send buffer, the NetClient control will attempt to flush it to the WinSock send buffers through send(). If the send buffer is then empty, or was empty initially empty, the Global buffer is reallocated.

Data Type

Integer

Scope

VB1.0+/VC++

See also

- RecvSize
- SendCount
- RecvCount
- RecvThreshold
- SendThreshold

SendThreshold Property, NetClient Control

See Also

Description

SendThreshold specifies the lowest number of characters allowed to be queued in the send buffer before the OnSend() event is fired.

Visual Basic

[*form .*] *NetClient*.**SendThreshold**[= *size%*]
[*size%*] = [*form .*] *NetClient*.**SendThreshold**

Visual C++

pNetClient->**GetNumProperty("SendThreshold")**
pNetClient->**SetNumProperty("SendThreshold",** *size%***)**

Remarks

By default, this property is set to 0. This means that when the send buffer empties, an OnSend() event will be fired.

This property can be set to any value in order to allow your application to maintain a full streaming connection.

This property is of enormous use to those programs that send large *chunks* of data. If your application is a bandwidth glutton, you will want to set this property to something akin to the initial value of SendSize (which, by default, returns the size from a getsockopt() with SO_SNDBUF). This will insure that your application keeps the underlying WinSock.DLL busy sending data.

Note: You MUST either use this property and the OnSend() event, or you must poll SendCount in a `DoEvents` style loop in order to send more data than SendSize bytes. If you do not, the NetClient control will trigger a WSAERR_SendOverflow(20004) VB error as soon as you overflow your WINSOCK.DLL's buffer and the NetClient control's send buffer.

WinSock Calls

None.

Data Type

Integer

Scope

VB1.0+/VC++

See also

- RecvThreshold
- SendSize
- RecvSize
- SendCount
- RecvCount

Socket Property, NetClient Control

Description

Socket specifies the actual socket number from a valid WinSock connection.

Visual Basic

[*form .*] *NetClient*.**Socket**=[*socket%*]
[*size%*] = [*form .*] *NetClient*.**Socket**

Visual C++

pNetClient->**GetNumProperty("Socket")**
pNetClient->**SetNumProperty("Socket",** *socket%***)**

Remarks

If Connect is False, this property has no meaning when read. The socket() call is not made until the Connect property is set.

If Connect is False, and you set this property to either 0 or SOCKET_ERROR (-1), the Connect property will be set to False - regardless.

During the period of setting Connect to True, and an OnConnect() or OnError() event occuring, the **Socket** property *IS* valid, albeit not connected.

If Connect is True, this property holds the return value of a socket() call - which, due to the fact that it is connected, is bound on both sides and ready for communication.

Whenever you set the **Socket** property with a valid WinSock Socket, Connect will be set to True, and getpeername() will be used to try and set HostAddr. The WSAAsyncSelect() call is also used to set up asyncronous message processing for the Socket, which can break older WinSocks from certain vendors (as they have assumed you will call WSAAsyncSelect() only once).

When you use the NetServer control to listen for incoming connections, you MUST use this property in it's OnAccept() event in order to start a connection.

Do *NOT* attempt a closesocket() on this handle. It will not currently have any adverse affects (other than sporadic errors), but it might in future versions. Do *NOT* set this **Socket** to blocking, and do *NOT* do a WSAAsyncSelect() on **Socket**, as it will surely break the current control.

For further reference, you may want to consult What is a Socket? and the WinSock.HLP file.

WinSock Calls

None.

Data Type

Integer

Scope

VB1.0+/VC++

TimeOut Property, NetClient Control

Description

TimeOut specifies the timeout period (in seconds) for starting the OnTimeOut() event.

Visual Basic

[*form .*] *NetClient*.**TimeOut**=[*seconds%*]
[*seconds%*] = [*form .*] *NetClient*.**TimeOut**

Visual C++

pNetClient->**GetNumProperty("TimeOut")**
pNetClient->**SetNumProperty("TimeOut",** *seconds%***)**

Remarks

Setting this property to a non zero value sets the number of seconds, from the time of setting, before the OnTimeOut() event is fired.

Setting this property to zero will *disable* the pending OnTimeOut() event.

Reading the **TimeOut** property at any time will tell you how many more seconds *remain* until the timeout will occur.

This property and event pair emulate the functionality of the Timer control.

WinSock Calls

None.

Data Type

Integer

Scope

VB1.0+/VC++

Version Property, NetClient Control

Description

This property reports the current revision of the NetClient control. The current version of *THIS* WSANET.VBX distribution will return "v1.08alpha" as the version string.

Visual Basic

[*version$*] = [*form .*] *NetClient*.**Version**

Visual C++

pNetClient->**GetStrProperty("Version")**

Remarks

This property can *NOT* be set.

You can use this property at run time to warn the user that the application may, *or may not*, work with any newer version of the control. The NetClient control currently has 8 reserved properties that can be used before any incompatibilities will occur. Hopefully, no more than 8 additional VB1.0 properties will needed to make this control do everything WinSock can do. Visual Basic 2.0 properties can be added at the end of the properties list, so no future incompatibilities should occur.

This string is merely a way for programmers to deal with known incompatibilities of WSANET.VBX versions, and to report version information to the user. This is *NOT* the way for an application to realize the version of WinSock it is using - this version of WSANET does *NOT* support any version beside WinSock v1.1.

WinSock Calls

None.

Data Type

String (Hsz)

Scope

VB1.0+/VC++

See Also
OnConnect
OnClose
OnError
OnRecv
OnSend
OnTimeOut

Using Events

OnConnect Event, NetClient Control

See Also

Description

Generated whenever a connection is made to a remote host.

Visual Basic

Sub *NetClient*_**OnConnect(**[*Index* **As Integer**]**)**

Visual C++

Function Signature:
void *CMyDialog::On Outline OnConnect***(UINT, int, CWnd*, LPVOID lpParams)**

Parameter Usage:
None.

Remarks

The **OnConnect()** event is generated whenever a connection to a remote host is completed. This event is fired in response to the Connect property being set to True, and a connection to a remote host successfully completing.

This event will not be fired if a connection can not be made with the specified remote host. Instead, the OnError() event will be called with an appropriate WinSock error.

WinSock Events

This event is fired on response to an FD_CONNECT message.

OnClose Event, NetClient Control

See Also

Description

Generated whenever a connection to a remote host is lost.

Visual Basic

Sub *NetClient*_**OnClose(**[*Index* **As Integer**]**)**

Visual C++

Function Signature:
void *CMyDialog::On Outline OnClose***(UINT, int, CWnd*, LPVOID lpParams)**

Parameter Usage:
None.

Remarks

The **OnClose()** event is generated whenever a connection to a remote host is lost.

This event is *NOT* fired due to your application setting the Connect property to False.

This event will *NOT* occur until after connection to a remote host is established.

WinSock Events

This event is fired on response to either an FD_CLOSE message or a 0 return by recv().

OnError Event, NetClient Control

See Also

Description

Generated whenever an error of some kind occurs.

Visual Basic

Sub *NetClient*_**OnError(**[*Index* **As Integer**], ErrorNumber **As Integer)**

Visual C++

Function Signature:
void *CMyDialog::On Outline OnError***(UINT, int, CWnd*, LPVOID lpParams)**

Parameter Usage:
None.

Remarks

The **OnError()** event is generated whenever an error occurs.

The **ErrorNumber** value and ErrorMessage property can be used to identify the error that occurred.

WinSock Events

This event is fired whenever an error occurs. WSAGetLastError() is called by the NetClient control if the error was WinSock related before the **OnError()** event is fired.

OnRecv Event, NetClient Control

See Also

Description

Generated whenever incoming data is received from a network connection.

Visual Basic

Sub *NetClient*_**OnRecv(**[*Index* **As Integer**]**)**

Visual C++

Function Signature:
void *CMyDialog::On Outline OnRecv***(UINT, int, CWnd*, LPVOID lpParams)**

Parameter Usage:
None.

Remarks

The **OnRecv()** event is generated whenever the count of incoming data rises above the RecvThreshold.

The RecvLine/Line/LineDelimiter properties can be used to parse incoming data that does not include embedded NULLs (Chr$(0))

The RecvBlock/Block properties can be used to parse incoming data that requires binary handling (in VB2.0+ only).

WinSock Events

This event is fired whenever incoming data is received. This data is received through the recv() WinSock call during the FD_READ message handler.

OnSend Event, NetClient Control

See Also

Description

Generated whenever data should be sent to the remote host.

Visual Basic

Sub *NetClient*_**OnSend(**[*Index* **As Integer**]**)**

Visual C++

Function Signature:
void *CMyDialog::On Outline OnSend***(UINT, int, CWnd*, LPVOID lpParams)**

Parameter Usage:
None.

Remarks

The **OnSend()** event is generated whenever the count of incoming data drops below the SendThreshold.

The SendLine and Line properties can be used to send outgoing data that does not include embedded NULLs (Chr$(0))

The SendBlock and Block properties can be used to send outgoing data that requires binary handling (in VB2.0+ only).

WinSock Events

This Event is typically the response to an FD_WRITE message that allowed the send buffer to drop below the SendThreshold.

OnTimeOut Event, NetClient Control

See Also

Description

Generated whenever the timeout period has expired.

Visual Basic

Sub *NetClient*_**OnSend(**[*Index* **As Integer**]**)**

Visual C++

Function Signature:
void *CMyDialog::On Outline OnTimeOut***(UINT, int, CWnd*, LPVOID lpParams)**

Parameter Usage:
None.

Remarks

The **OnTimeOut()** event is generated whenever the desired timer period has elapsed.

By setting the TimeOut property to a non-zero value, your application is setting the number of seconds before the **OnTimeOut()** event occurs.

This event the response to the TimeOut property reaching 0. This event is one-shot only; after this event occurs, it will not occur again unless the TimeOut property is set again.

WinSock Events

None.

NetClientSendBlock() Export, NetClient Control

See Also

Description

This function will send any Visual Basic string as binary to a remote host.

Visual Basic

Declare Function **NetClientSendBlock** Lib "WSANET.VBX" (cControl As NetClient, sString As String) As Integer

Visual C++

int CALLBACK **NetClientSendBlock**(HCTL, HLSTR);

NOTE: I do not know the declarations in order to let VC++ pass a VB string to a function.

Remarks

This function supports the transmission of binary data through the NetClient control. This procedure is exported as a C style *function* that accepts a Visual Basic string as a parameter (note the absence of the ByVal keyword).

If you are able to use VB2.0+ HLSTR properties, then you will probably want to use the Block series of properties.

If you need to receive binary data through VB1.0 or VC++, you will need to use the NetClientRecvBlock() function.

WinSock Calls

send()

Scope

VB1.0+/VC++

See Also
NetClientRecvBlock
Block
SendBlock
RecvBlock

NetClientRecvBlock() Export, NetClient Control

See Also

Description

This function will receive any Visual Basic string as binary from a remote host.

Visual Basic

Declare Function **NetClientRecvBlock** Lib "WSANET.VBX" (cControl As NetClient, sString As String, iNumber As Integer) As Integer

Visual C++

int CALLBACK **NetClientRecvBlock**(HCTL, HLSTR, int);

NOTE: I do not know the declarations in order to let VC++ pass a VB string to a function.

Remarks

This function supports the transmission of binary data through the NetClient control. This procedure is exported as a C style *function* that accepts a Visual Basic string as a parameter (note the absence of the ByVal keyword).

If you are able to use VB2.0+ HLSTR properties, then you will probably want to use the Block series of properties.

If you need to send binary data through VB1.0 or VC++, you will need to use the NetClientSendBlock() function.

WinSock Calls

This function doesn't use any WinSock routines. Anything that this function returns is merely buffered data that *WAS* received by the recv() function call in the FD_READ message handler for the control.

Scope

VB1.0+/VC++

See Also
NetClientSendBlock
Block
SendBlock
RecvBlock