Windows* Sockets 2 Application Programming Interface

An Interface for Transparent Network Programming
Under Microsoft Windows™

Revision 2.2.2
August 7, 1997

Subject to Change Without Notice

Disclaimer and License

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

A LICENSE IS HEREBY GRANTED TO REPRODUCE THIS SPECIFICATION, BUT ONLY IN ITS ENTIRETY AND WITHOUT MODIFICATION. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY OTHER INTELLECTUAL PROPERTY RIGHTS IS GRANTED HEREIN.

INTEL, MICROSOFT, STARDUST, AND THE OTHER COMPANIES WHOSE CONTRIBUTIONS ARE ACKNOWLEDGED BELOW DISCLAIM ALL LIABILITY, INCLUDING LIABILITY FOR INFRINGEMENT OF ANY PROPRIETARY RIGHTS, RELATING TO IMPLEMENTATION OF INFORMATION IN THIS SPECIFICATION. SAID COMPANIES DO NOT WARRANT OR REPRESENT THAT SUCH IMPLEMENTATION(S) WILL NOT INFRINGE SUCH RIGHTS.

* Third-party trademarks are the property of their respective owners.

Table of Contents

Acknowledgments
Windows Sockets Version 2

Since The WinSock Group started the Version 2 specification process in May 1994, hundreds of people, companies and organizations have cooperated and contributed to its design and specification. Several meetings, many emails and telephone conversations later, it's appropriate to acknowledge the part played by everyone and certain contributors in particular.

Many individuals too numerous to mention have given time to the project and all of them are owed a debt of thanks for the roles they played in creating the most comprehensive open transport API designed to date. The commitment, dedication and energy of the following individuals and companies should be singled out for special attention.

First, the design of WinSock 2 was based on the input of multiple "Functionality Groups" whose leaders cajoled, steered, defined and refined each of their group's technical proposals. Consequently, we'd like to recognize the following individuals and their employers for the time and effort they have given. It's appropriate to thank Dave Andersen for the challenge he undertook, met and surpassed in defining the generic API set and pulling together the contributions of all the various Functionality Groups.

Functionality Group	**Leader(s)**	**Email**	**Company**
Generic API	Dave Andersen	andersen@ibeam.jf.intel.com	Intel
Operating Framework	Keith Moore	keithmo@microsoft.com	Microsoft
Specification Clarifications	Bob Quinn Vikas Garg Paul Brooks	rcq@sockets.com vikas@distinct.com paul@turbosoft.com.au	sockets.com Distinct Turbosoft
Name Resolution	Margaret Johnson Cameron Ferroni Paul Drews	margretj@microsoft.com cameronf@microsoft.com Paul_C_Drews@ccm.jf.intel.com	Microsoft Microsoft Intel
Connection-Oriented Media	Charlie Tai Sanjay Agrawal Kumar	Charlie_Tai@ccm.jf.intel.com kumar@fvc.com	Intel Microsoft
Wireless	Dale Buchholz	drbuchholz@mot.com	Motorola
TCP/IP	Michael Khalandovsky	mlk@ftp.com	FTP Software
IPX/SPX	Tim Delaney	tdelaney@novell.com	Novell
DECnet	Cathy Bence	bence@ranger.enet.dec.com	DEC
OSI	Adrian Dawson	ald@oasis.icl.co.uk	ICL

The following individuals moderated the WinSock 2 effort as a whole and provided the framework, technical guidance and administrative mechanisms for WinSock Version 2.

Moderator	**Email**	**Company**
Martin Hall	martinh@stardust.com	Stardust Technologies
Dave Treadwell	davidtr@microsoft.com	Microsoft
Mark Towfiq	towfiq@east.sun.com	SunSoft

Special thanks to Microsoft and Intel for the amount of time these companies gave to the specification and especially to Dave Treadwell and Keith Moore at Microsoft and Dave Andersen and Charlie Tai at Intel for their considerable editorial efforts on the WinSock 2 specifications.

The SDK for Windows NT and Windows 95 was a project in its own right and was brought about by a joint effort between Microsoft and the Intel Architecture Labs. The Microsoft team included Dave Treadwell, Steve Firebaugh, Keith Moore, Arnold Miller, Francis X. Langlois, Mosin Ahmed, Chris Steck and Dave Beaver. The Intel team included Dave Andersen, Dave Doerner, Paul Drews, Charlie Tai, Dirk Brandewie, Dan Chou, Michael Grafton and Dan Ohlemacher.

This version would not, of course, have been possible without the effort of the contributors to WinSock Version 1.1 and the numerous products that implement and use it. Of special significance to the success of WinSock are the hundreds of shareware and freeware applications that have been developed and continue to emerge. The authors of these packages are some of WinSock's unsung heroes. It's fitting to recognize, at least, the role of and contribution made by Peter Tattam's "Trumpet" WinSock implementation.

We'd like to thank Interop for hosting the kick-off meeting for WinSock Version 2, and Novell for kindly providing the facilities for the meeting that marked the consolidation effort which brought together the work of different groups into a coordinated API and SPI definition.

Sincerely,

Martin Hall
Stardust Technologies

1. Introduction

This document specifies the Windows Sockets 2 programming interface. Windows Sockets 2 utilizes the sockets paradigm as first popularized in BSD UNIX[1] and as adapted for Microsoft Windows in the Windows Sockets 1.1 specification. While historically sockets programming in general and Windows Sockets programming in particular has been TCP/IP-centric, this is no longer the case. Consequently, it is important to realize that Windows Sockets 2 is an **interface** and not a **protocol**. As an interface it is used to discover and utilize the communications capabilities of any number of underlying transport protocols. Because it is not a protocol, it does not in any way affect the "bits on the wire", and does not need to be utilized on both ends of a communications link.

The motivation in creating version 2 of Windows Sockets was primarily to provide a protocol-independent transport interface that is fully capable of supporting emerging networking capabilities including real-time multimedia communications. Thus Windows Sockets 2 is a true superset of the widely deployed Windows Sockets 1.1 interface. While maintaining full backwards compatibility it extends the Windows Sockets interface in a number of areas including

- Access to protocols other than TCP/IP: WinSock 2 allows an application to use the familiar socket interface to achieve simultaneous access to any number of installed transport protocols.
- Protocol-independent name resolution facilities: WinSock 2 includes a standardized set of APIs for querying and working with the myriad of name resolution domains that exist today (e.g. DNS, SAP, X.500, etc.)
- Overlapped I/O with scatter/gather: following the model established in Win32 environments, WinSock 2 incorporates the overlapped paradigm for socket I/O and incorporates scatter/gather capabilities as well.
- Quality of service: WinSock 2 establishes conventions for applications to negotiate required service levels for parameters such as bandwidth and latency. Other QOS-related enhancements include socket grouping and prioritization, and mechanisms for network-specific QOS extensions.
- Protocol-independent multicast and multipoint: applications can discover what type of multipoint or multicast capabilities a transport provides and use these facilities in a generic manner.
- Other frequently requested extensions: shared sockets, conditional acceptance, exchange of user data at connection setup/teardown time, protocol-specific extension mechanisms.

It is almost always the case that adding more capability and functionality also increases the level of complexity, and Windows Sockets 2 is no exception. We have attempted to mitigate this as much as possible by extending the familiar sockets interface as opposed to starting from scratch. While this approach offers significant benefit to experienced socket programmers, it may occasionally vex them as well since certain widely held assumptions are no longer valid. For example, many sockets programmers assume that all connectionless protocols use SOCK_DGRAM sockets and all connection-oriented protocols use SOCK_STREAM sockets. In WinSock 2 SOCK_DGRAM and SOCK_STREAM are only two of many possible socket types, and programmers should no longer rely on socket type to describe all of the essential attributes of a transport protocol.

With its vastly larger scope, Windows Sockets 2 takes the socket paradigm considerably beyond what it's original designers contemplated. As a consequence, a number of new functions have been added, all of which are assigned names that are prefixed with "WSA". In all but a few instances these new functions are expanded versions of an existing function from BSD sockets. The need to retain backwards compatibility mandates that we retain both the "just plain" BSD functions and the new "WSA" versions, which in turn amplifies the perception of WinSock 2 as being large and complex. This stretching of the sockets paradigm also requires us to occasionally dance around areas where the original sockets

[1] UNIX is a trademark of UNIX Systems Laboratories, Inc.

architecture is on shaky ground. A telling example of this is the unfortunate, but now irrevocable, decision to combine the notions of address family and protocol family.

1.1. Intended Audience

This document is targeted at persons who are familiar with the sockets network programming paradigm in general and, to a lesser degree, the Windows Sockets version 1.1 interface in particular. For an introduction to WinSock programming please refer to one or more of the excellent references cited in *Appendix D. For Further Reference.*.

Persons who are interested in developing applications that will take advantage of WinSock 2's capabilities will be primarily interested in this API specification. Persons who are interested in making a particular transport protocol available under the WinSock 2 interface will need to be familiar with the WinSock 2 Service Provider Interface (SPI) Specification as well. The Windows Sockets 2 SPI specification exists under separate cover.

1.2. Document Organization

The complete Windows Sockets 2 specification consists of four separate documents:

1. ***Windows Sockets 2 Application Programming Interface***
2. ***Windows Sockets 2 Protocol-Specific Annex***
3. ***Windows Sockets 2 Service Provider Interface***
4. ***Windows Sockets 2 Debug-Trace DLL***

This document (***Windows Sockets 2 Application Programming Interface***) is divided into four main sections and four appendices.

Section 1	Introductory material about the specification as a whole
Section 2	Summary of additions and changes in going from Windows Sockets 1.1 to Windows Sockets 2
Section 3	Windows Sockets Programming Considerations
Section 4	Detailed reference information for the data transport functions
Section 5	Introductory material and detailed reference information for the name resolution and registration functions
Appendix A	Information on WinSock header files, error codes and data type definitions
Appendix B	Details on multipoint and multicast features in Windows Sockets 2
Appendix C	The WinSock Lame List
Appendix D	A bibliography for those seeking additional information

The ***Windows Sockets 2 Protocol-Specific Annex*** contains information specific to a number of transport protocols that are accessible via Windows Sockets 2. The ***Windows Sockets 2 Service Provider Interface*** specifies the interface that transport providers must conform to in order to be accessible via Windows Sockets 2. ***Windows Sockets 2 Debug-Trace DLL*** describes the files and mechanics of the debug-trace DLL. This is a useful tool for application developers and service providers alike, that shows API and SPI calls in and out of the WinSock 2 DLL..

1.3. Status of This Specification

Version 2.2.1 of the API specification is considered final with respect to functionality. Future revisions of this specification are contemplated only for the purpose of correcting errors or removing ambiguity, not as a means of incorporating additional functionality.

This document comprises only the API portion of the Windows Sockets 2 specification. The WinSock Group's Generic API Extensions functionality group produced the initial versions of this document as

well as the ***Windows Sockets 2 Protocol-Specific Annex***. Constructive comments and feedback on this material are always welcome and should be directed towards:

David B. Andersen
Intel Architecture Labs
andersen@ibeam.jf.intel.com

1.4. Document Version Conventions

Starting with draft release 2.0.6, the API and SPI documents have adopted a 3-part revision identification system. Each revision of the document will be clearly labeled with a release date and a revision identifier such as *X.Y.Z* where:

X is the *major version* of the WinSock specification (currently version 2)

Y is a *major revision* identifier that is incremented each time changes are made that impact binary compatibility with the previous spec revision (e.g. changes in a function's parameter list or new functions being added)

Z is a *minor revision* indicator that is incremented when wording changes or clarifications have been made which do not impact binary compatibility with a previous revision.

Note that gaps in the minor revision indicator (*Z*) between successive releases of a document are not unusual, especially during the early stages of a document's life when many changes are occurring.

1.5. New And/Or Different in Version 2.2.1

Version 2.2.1 is being released primarily to correct errors and omissions from earlier versions of the specifications.

1.6. New And/Or Different in Version 2.2.2

Version 2.2.2 is being released to add new functionality for querying and receiving notification of changes in network and system configuration. This new functionality consists of one new function (WSAProviderConfigChange()) four new socket IOCTLs (**SIO_ROUTING_INTERFACE_QUERY**, **SIO_ROUTING_INTERFACE_CHANGE**, **SIO_ADDRESS_LIST_QUERY**, **SIO_ADDRESS_LIST_CHANGE**), and two new asynchronous network events (**FD_ROUTING_INTERFACE_CHANGE** and **FD_ADDRESS_LIST_CHANGE**) reported via existing functions (**WSAAsyncSelect()**,**WSAEventSelect(),** and **WSAEnumNetworkEvents()**).

2. Summary of New Concepts, Additions and Changes for WinSock 2

The paragraphs that follow summarize the completely new architecture for WinSock 2 and describe the major changes and additions in going from WinSock 1.1 to WinSock 2. For detailed information about how to use a specific function or feature, please refer to the appropriate API description(s) in sections 3 or 4.

2.1. WinSock 2 Architecture

A number of the features incorporated into WinSock 2 required that a substantial change in architecture occur. Foremost among these is the ability of an application to access multiple different transport protocols simultaneously. Other factors included the adoption of protocol-independent name resolution facilities, provisions for layered protocols and protocol chains, and a different mechanism for WinSock service providers to offer extended, provider-specific functionality.

2.1.1. Simultaneous Access to Multiple Transport Protocols

In order to provide simultaneous access to multiple transport protocols, the WinSock architecture has changed in Version 2. With WinSock Version 1.1, the DLL which implements the WinSock interface is supplied by the vendor of the TCP/IP protocol stack. The interface between the WinSock DLL and the underlying stack was both unique and proprietary. WinSock 2 changes the model by defining a standard service provider interface (SPI) between the WinSock DLL and protocol stacks. This makes it possible for multiple stacks from multiple vendors to be accessed simultaneously from a single WinSock DLL. Furthermore, WinSock 2 support is not limited to TCP/IP protocol stacks as is the case for WinSock 1.1. The Windows Open System Architecture (WOSA) compliant WinSock 2 architecture is illustrated as follows:

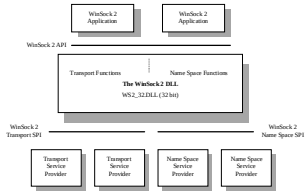

Figure 1 WinSock 2 Architecture

Note: for simplicity's sake the WS2_32.DLL will be referred to simply as WinSock 2 DLL.

With the above architecture, it is no longer necessary (or desirable) for each stack vendor to supply their own implementation of WinSock 2 DLL, since a single WinSock 2 DLL must work across all stacks. The WinSock 2 DLL and compatibility shims should thus be viewed in the same light as an operating system component.

2.1.2. Backwards Compatibility For WinSock 1.1 Applications

WinSock Version 2 has been made backward compatible with WinSock Version 1.1 at 2 levels: source and binary. This facilitates maximum interoperability between WinSock apps of any version and WinSock implementations of any version, and minimizes pain and confusion for users of WinSock apps, network stacks and service providers. Note that current WinSock 1.1 compliant applications are guaranteed to run over a WinSock 2 implementation without modification of any kind as long as at least one TCP/IP service provider is properly installed.

2.1.2.1. Source Code Compatibility

Source code compatibility in WinSock Version 2 means that with few exceptions, all the WinSock Version 1.1 API's are preserved in WinSock Version 2. WinSock 1.1 applications that make use of

blocking hooks will need to be modified as blocking hooks are no longer supported in WinSock 2. See section *3.3.2. Winsock 1.1 Blocking routines & EINPROGRESS* for more information. This means that existing WinSock 1.1 application source code can easily be moved to the WinSock 2 system at a simple level by including the new header file "winsock2.h" and performing a straightforward re-link with the appropriate WinSock 2 libraries. Application developers are encouraged to view this as merely the first step in a full transition to WinSock Version 2. This is because there are numerous ways in which a WinSock 1.1 application can be improved by exploring and using functionality that is new in WinSock Version 2.

2.1.2.2. Binary Compatibility

A major design goal for WinSock Version 2 was enabling existing WinSock Version 1.1 applications to work unchanged at a binary level with WinSock Version 2. Since WinSock 1.1 applications were TCP/IP-based, binary compatibility implies that TCP/IP-based WinSock 2 Transport and Name Resolution Service Providers are present in the WinSock 2 system. In order to enable WinSock Version 1.1 applications in this scenario, the WinSock Version 2 system has an additional "shim" component supplied with it: a Version 1.1 compliant WINSOCK DLL. Installation guidelines for the WinSock 2 system ensure that the introduction of the WinSock 2 components to an end user system has no negative impact on users' existing WinSock-based applications.

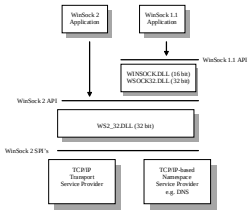

Figure 2 WinSock 1.1 Compatibility Architecture

WinSock 1.1 applications currently use certain elements from the **WSAData** structure (obtained via a call to **WSAStartup()**) to obtain information about the underlying TCP/IP stack.. These include: *iMaxSockets, iMaxUdpDg*, and *lpVendorInfo*. While WinSock 2 applications will know to ignore these values (since they cannot uniformly apply to all available protocol stacks), safe values are supplied to avoid breaking WinSock 1.1 applications.

2.2. Making Transport Protocols Available To WinSock

In order for a transport protocol to be accessible via WinSock it must be properly installed on the system and registered with WinSock. The WinSock 2 DLL exports a set of APIs to facilitate the registration process. These include creating a new registration and removing an existing one. When new registrations are created the caller (presumed to be the stack vendor's installation script) supplies one or more filled in WSAPROTOCOL_INFO structs which contain a complete set of information about the protocol. Please refer to the SPI document for details on how this is accomplished.

Note that transport stacks that are thus installed are considered to be WinSock service providers, and hereafter are referred to as such. The WinSock 2 SDK includes a small Windows applet called SPORDER.EXE that will allow the user to view and modify the order in which service providers are enumerated. Using this applet, a user may manually establish a particular TCP/IP protocol stack as the default TCP/IP provider if more than one such stack is present.

The functions that this applet uses to reorder the service providers are exported from a DLL (SPORDER.DLL). As a result, installation applications may use the interface of SPORDER.DLL to programmatically reorder service providers, to suit their needs. Please refer to the Service Provider Interface document for a detailed description of the interface.

2.2.1. Layered Protocols and Protocol Chains

WinSock 2 accommodates the notion of a layered protocol. A layered protocol is one that implements only higher level communications functions, while relying on an underlying transport stack for the actual exchange of data with a remote endpoint. An example of such a layered protocol would be a security layer that adds protocol to the connection establishment process in order to perform authentication and to establish a mutually agreed upon encryption scheme. Such a security protocol would generally require the services of an underlying reliable transport protocol such as TCP or SPX. The term *base protocol* refers to a protocol such as TCP or SPX which is fully capable of performing data communications with a remote endpoint, and the term *layered protocol* is used to describe a protocol that cannot stand alone. A *protocol chain* is defined as one or more layered protocols strung together and anchored by a base protocol.

This stringing together of one or more layered protocols and a base protocol into chains can be accomplished by arranging for the layered protocols to support the WinSock 2 SPI at both their upper and lower edges. A special WSAPROTOCOL_INFO struct is created which refers to the protocol chain as a whole, and which describes the explicit order in which the layered protocols are joined. This is illustrated in Figure 3. Note that since only base protocols and protocol chains are directly usable by applications, only these protocols are listed when the installed protocols are enumerated with **WSAEnumProtocols()**.

Figure 3 Layered Protocol Architecture

2.2.2. Using Multiple Protocols

An application may use **WSAEnumProtocols()** to discover which transport protocols and protocol chains are present and obtain information about each as contained in the associated WSAPROTOCOL_INFO struct. In most instances, there will be a single WSAPROTOCOL_INFO struct for each protocol or protocol chain. Some protocols however, are able to exhibit multiple behaviors. For example the SPX protocol is message-oriented (i.e. the sender's message boundaries are preserved by the network), but the receiving end may choose to ignore these message boundaries and treat the socket as a byte stream. Thus there could reasonably be two different WSAPROTOCOL_INFO struct entries for SPX, one for each of these behaviors.

Whereas in WinSock 1.1 there is a single address family (AF_INET) comprising a small number of well-known socket types and protocol identifiers, the focus will shift for WinSock 2. The existing address family, socket type and protocol identifiers are retained for compatibility reasons, but many new address family, socket type and protocol values will appear which are unique but not necessarily well known. Not being well known need not pose a problem since applications that desire to be protocol-independent are encouraged to select protocols for use on the basis of their suitability rather than the particular values assigned to their *socket_type* or *protocol* fields. Protocol suitability is indicated by the communications attributes (e.g. message vs. byte-stream oriented, reliable vs. unreliable, etc.) contained within the protocol's WSAPROTOCOL_INFO struct. Selecting protocols on the basis of suitability as opposed to well-known protocol names and socket types allows protocol-independent applications to take advantage of new transport protocols and their associated media types as they become available.

In terms of the well-known client/server paradigm, the server half of a client/server application will benefit by establishing listening sockets on all suitable transport protocols. The client, then, may establish its connection using any suitable protocol. This would enable, for example, a client application to be unmodified whether it was running on a desktop system connected via LAN or on a laptop using a wireless network.

A WinSock 2 clearinghouse has been established for protocol stack vendors to obtain unique identifiers for new address families, socket types and protocols. FTP and world-wide web servers are used to supply current identifier/value mappings, and email is used to request allocation of new ones. At the time of this writing the world-wide web URL for the Windows Sockets 2 Identifier Clearinghouse is

```
http://www.stardust.com/wsresource/winsock2/ws2ident.html
```

2.2.3. Multiple Provider Restrictions on select()

In WinSock 2 the FD_SET supplied to the **select()** function is constrained to contain sockets associated with a single service provider. This does not in any way restrict an application from having multiple sockets open using multiple providers. When non-blocking operations are preferred the **WSAAsyncSelect()** function is the solution. Since it takes a socket descriptor as an input parameter, it doesn't matter what provider is associated with the socket. When an application needs to use blocking semantics on a set of sockets that spans multiple providers, the recommended solution is to use **WSAWaitForMultipleEvents()**. The application may also choose to take advantage of the **WSAEventSelect()** function which allows the FD_XXX network events to be associated with an event object and handled from within the event object paradigm (described below).

2.3. Function Extension Mechanism

Since the WinSock DLL itself is no longer supplied by each individual stack vendor, it is no longer possible for a stack vendor to offer extended functionality by just adding entry points to the WinSock DLL. To overcome this limitation, WinSock 2 takes advantage of the new **WSAIoctl()** function to accommodate service providers who wish to offer provider-specific functionality extensions. This mechanism presupposes, of course, that an application is aware of a particular extension and understands

both the semantics and syntax involved. Such information would typically be supplied by the service provider vendor.

In order to invoke an extension function, the application must first ask for a pointer to the desired function. This is done via the **WSAIoctl()** function using the SIO_GET_EXTENSION_FUNCTION_POINTER command code. The input buffer to the **WSAIoctl()** function contains an identifier for the desired extension function and the output buffer will contain the function pointer itself. The application can then invoke the extension function directly without passing through the WinSock 2 DLL.

The identifiers assigned to extension functions are globally unique identifiers (GUIDs) that are allocated by service provider vendors.. Vendors who create extension functions are urged to publish full details about the function including the syntax of the function prototype. This makes it possible for common and/or popular extension functions to be offered by more than one service provider vendor. An application can obtain the function pointer and use the function without needing to know anything about the particular service provider that implements the function.

2.4. Debug and Trace Facilities

When the developer of a WinSock 2 application encounters a WinSock-related bug there is a need to isolate the bug to either (1) the application, (2) the WinSock 2 DLL (or one of it's 1.1 compatibility "shim" DLLs), or (3) the service provider. WinSock 2 addresses this need through a specially instrumented version of the WinSock 2 DLL and a separate debug/trace DLL. This combination allows all procedure calls across the WinSock 2 API or SPI to be monitored, and to some extent controlled.

Developers can use this mechanism to trace procedure calls, procedure returns, parameter values, and return values. Parameter values and return values can be altered on procedure-call or procedure-return. If desired, a procedure-call can even be prevented or redirected. With access to this level of information and control, it should be easy for a developer to isolate any problem to the application, WinSock 2 DLL or service provider.

The WinSock 2 SDK includes the instrumented WinSock 2 DLL, a sample debug/trace DLL and a document containing a detailed description of the above two components. The sample debug/trace DLL is provided in both source and object form. Developers are free to use the source to develop versions of the debug/trace DLL that meet their special needs.

2.5. Protocol Independent Name Resolution

WinSock 2 includes provisions for standardizing the way applications access and use the various network name resolution services. WinSock 2 applications will not need to be cognizant of the widely differing interfaces associated with name services such as DNS, NIS, X.500, SAP, etc. An introduction to this topic and the details of the APIs are currently located in section *5.1. Protocol-Independent Name Resolution*

2.6. Overlapped I/O and Event Objects

WinSock 2 introduces overlapped I/O and requires that all transport providers support this capability. Overlapped I/O can be performed only on sockets that were created via the **WSASocket()** function with the WSA_FLAG_OVERLAPPED flag set (or created via the **socket()** function), and follows the model established in Win32.

Note that creating a socket with the overlapped attribute has no impact on whether a socket is currently in the blocking or non-blocking mode. Sockets created with the overlapped attribute may be used to

perform overlapped I/O, and doing so does not change the blocking mode of a socket. Since overlapped I/O operations do not block, the blocking mode of a socket is irrelevant for these operations.

For receiving, applications use **WSARecv()** or **WSARecvFrom()** to supply buffers into which data is to be received. If one or more buffers are posted prior to the time when data has been received by the network, it is possible that data will be placed into the user's buffers immediately as it arrives and thereby avoid the copy operation that would otherwise occur at the time the **recv()** or **recvfrom()** function is invoked. If data is already present when receive buffers are posted, it is copied immediately into the user's buffers. If data arrives when no receive buffers have been posted by the application, the network resorts to the familiar synchronous style of operation where the incoming data is buffered internally until such time as the application issues a receive call and thereby supplies a buffer into which the data may be copied. An exception to this would be if the application used **setsockopt()** to set the size of the receive buffer to zero. In this instance, reliable protocols would only allow data to be received when application buffers had been posted, and data on unreliable protocols would be lost.

On the sending side, applications use **WSASend()** or **WSASendTo()** to supply pointers to filled buffers and then agree to not disturb the buffers in any way until such time as the network has consumed the buffer's contents.

Overlapped send and receive calls return immediately. A return value of zero indicates that the I/O operation completed immediately and that the corresponding completion indication has already occurred. That is, the associated event object has been signaled, or a completion routine has been queued and will be executed when the calling thread gets into the alterable wait state. A return value of SOCKET_ERROR coupled with an error code of WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that a subsequent indication will be provided when send buffers have been consumed or when a receive operation has been completed. However, for byte stream style sockets, the completion indication occurs whenever the incoming data is exhausted, regardless of whether the buffers are fully filled. Any other error code indicates that the overlapped operation was not successfully initiated and that no completion indication will be forthcoming.

Both send and receive operations can be overlapped. The receive functions may be invoked multiple times to post receive buffers in preparation for incoming data, and the send functions may be invoked multiple times to queue up multiple buffers to be sent. Note that while the application can rely upon a series of overlapped send buffers being sent in the order supplied, the corresponding completion indications may occur in a different order. Likewise, on the receiving side, buffers will be filled in the order they are supplied but the completion indications may occur in a different order.

There is no way to cancel individual overlapped operations pending on a given socket, however, the **closesocket()** function can be called to close the socket and eventually discontinue all pending operations.

The deferred completion feature of overlapped I/O is also available for **WSAIoctl()** which is an enhanced version of **ioctlsocket()**.

2.6.1. Event Objects

Introducing overlapped I/O requires a mechanism for applications to unambiguously associate send and receive requests with their subsequent completion indications. In WinSock 2 this may be accomplished via event objects which are modeled after Win32 events. WinSock event objects are fairly simple constructs which can be created and closed, set and cleared, waited upon and polled. Their prime usefulness comes from the ability of an application to block and wait until one or more event objects become set.

Applications use **WSACreateEvent()** to obtain an event object handle which may then be supplied as a required parameter to the overlapped versions of send and receive calls (**WSASend(), WSASendTo(), WSARecv(), WSARecvFrom()**). The event object, which is cleared when first created, is set by the transport providers when the associated overlapped I/O operation has completed (either successfully or with errors). Each event object created by **WSACreateEvent()** should have a matching **WSACloseEvent()** to destroy it.

Event objects are also used in **WSAEventSelect()** to associate one or more FD_XXX network events with an event object. This is described in section 2.7. Asynchronous Notification Using Event Objects.

In 32-bit environments, event object related functions, including **WSACreateEvent()**, **WSACloseEvent()**, **WSASetEvent()**, **WSAResetEvent()**, and **WSAWaitForMultipleEvents()**are directly mapped to the corresponding native Win32 functions, i.e. the same function name without the **WSA** prefix.

2.6.2. Receiving Completion Indications

In order to provide applications with appropriate levels of flexibility, several options are available for receiving completion indications. These include: waiting on (i.e. blocking on) event objects, polling event objects, and socket I/O completion routines.

2.6.2.1. Blocking and Waiting for Completion Indication

Applications may choose to block while waiting for one or more event objects to become set using **WSAWaitForMultipleEvents()**. Since WinSock 2 event objects are implemented as Win32 events, the native Win32 function **WaitForMultipleObjects()** may also be used for this purpose. This is especially useful if the thread needs to wait on both socket and non-socket events.

2.6.2.2. Polling for Completion Indication

Applications that prefer not to block may use **WSAGetOverlappedResult()** to poll for the completion status associated with any particular event object. This function indicates whether or not the overlapped operation has completed, and, if completed, arranges for **WSAGetLastError()** to retrieve the error status of the overlapped operation.

2.6.2.3. Using socket I/O completion routines

The functions used to initiate overlapped I/O (**WSASend(), WSASendTo(), WSARecv(), WSARecvFrom()**) all take *lpCompletionRoutine* as an optional input parameter. This is a pointer to an application-specified function that will be called after a successfully initiated overlapped I/O operation has completed (successfully or otherwise). The completion routine follows the same rules as stipulated for Win32 file I/O completion routines. The completion routine will not be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents()** is invoked with the *fAlertable* flag set. Note that an application that uses the completion routine option for a particular overlapped I/O request may not use the "wait" option of **WSAGetOverlappedResult()** for that same overlapped I/O request.

Transports allow an application to invoke send and receive operations from within the context of the socket I/O completion routine, and guarantee that, for a given socket, I/O completion routines will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

2.6.2.4. Summary of overlapped completion indication mechanisms

The particular overlapped I/O completion indication to be used for a given overlapped operation is determined by whether or not the application supplies a pointer to a completion function, whether or not a WSAOVERLAPPED structure is referenced, and the value of the *hEvent* field within the WSAOVERLAPPED structure (if supplied). The following table summarizes the completion semantics

for an overlapped socket, showing the various combination of *lpOverlapped*, *hEvent*, and *lpCompletionRoutine*:

lpOverlapped	hEvent	lpCompletionRoutine	Completion Indication
NULL	not applicable	ignored	Operation completes synchronously, i.e. it behaves as if it were a non-overlapped socket.
!NULL	NULL	NULL	Operation completes overlapped, but there is no WinSock 2 supported completion mechanism. The completion port mechanism (if supported) may be used in this case, otherwise there will be no completion notification.
!NULL	!NULL	NULL	Operation completes overlapped, notification by signaling event object.
!NULL	ignored	!NULL	Operation completes overlapped, notification by scheduling completion routine.

2.6.3. WSAOVERLAPPED Details

The **WSAOVERLAPPED** structure provides a communication medium between the initiation of an overlapped I/O operation and its subsequent completion. The **WSAOVERLAPPED** structure is designed to be compatible with the Win32 **OVERLAPPED** structure:

```
typedef struct _WSAOVERLAPPED {
      DWORD       Internal;         // reserved
      DWORD       InternalHigh;     // reserved
      DWORD       Offset;           // reserved
      DWORD       OffsetHigh;       // reserved
      WSAEVENT    hEvent;
} WSAOVERLAPPED, LPWSAOVERLAPPED;
```

Internal — This reserved field is used internally by the entity that implements overlapped I/O. For service providers that create sockets as installable file system (IFS) handles, this field is used by the underlying operating system. Other service providers (non-IFS providers) are free to use this field as necessary.

InternalHigh — This reserved field is used internally by the entity that implements overlapped I/O. For service providers that create sockets as IFS handles, this field is used by the underlying operating system. Non-IFS providers are free to use this field as necessary.

Offset — This field is reserved for service providers to use.

OffsetHigh	This field is reserved for service providers to use.
hEvent	If an overlapped I/O operation is issued without an I/O completion routine (*lpCompletionRoutine* is NULL), then this field should either contain a valid handle to a **WSAEVENT** object or be NULL. If *lpCompletionRoutine* is non-NULL then applications are free to use this field as necessary. An application that uses a non-NULL *lpCompletionRoutine* for a particular overlapped I/O request may not use the "wait" option of **WSAGetOverlappedResult()** for that same overlapped I/O request.

2.7. Asynchronous Notification Using Event Objects

To accommodate applications such as daemons and services that have no user interface (and hence do not use Windows handles), the **WSAEventSelect()** and **WSAEnumNetworkEvents()** functions are provided. **WSAEventSelect()** behaves exactly like **WSAAsyncSelect()** except that, rather than cause a Windows message to be sent on the occurrence of an FD_XXX network event (e.g.. FD_READ, FD_WRITE, etc.), an application-designated event object is set.

Also, the fact that a particular FD_XXX network event has occurred is "remembered" by the service provider. The application may call **WSAEnumNetworkEvents()** to have the current contents of the network event memory copied to an application supplied buffer and to have the network event memory atomically cleared. If desired, the application may also designate a particular event object which is cleared along with the network event memory.

2.8. Quality of Service

The basic QOS mechanism in WinSock 2 descends from the flow specification (or "flow spec") as described by Craig Partridge in RFC 1363, dated September 1992. A brief overview of this concept is as follows:

Flow specs describe a set of characteristics about a proposed unidirectional flow through the network. An application may associate a pair of flowspecs with a socket (one for each direction) at the time a connection request is made using **WSAConnect()**, or at other times using **WSAIoctl()** with the SIO_SET_QOS/SIO_SET_GROUP_QOS command. Flowspecs indicate parametrically what level of service is required and provide a feedback mechanism for applications to use in adapting to network conditions.

The usage model for QOS in WinSock 2 is as follows. An application may establish its QOS requirements at any time via **WSAIoctl()** or coincident with the connect operation via **WSAConnect()**. For connection-oriented transports, it is often most convenient for an application to use **WSAConnect()**, and any QOS values supplied at connect time supersede those that may have been supplied earlier via **WSAIoctl()**. If the **WSAConnect()** function completes successfully the application knows that its QOS request has been honored by the network, and the application is then free to use the socket for data exchange. If the connect operation fails because of limited resources an appropriate error indication is given. At this point the application may scale down its service request and try again or simply give up.

After every connection attempt (successful or otherwise) transport providers update the associated flow spec structures in order to indicate, as well as possible, the existing network conditions. (Note that it is legal to update with the default values defined below to indicate that information about the current network conditions is not available.) This update from the service provider about current network conditions is especially useful for the case where the application's QOS request consisted entirely of the

default (i.e. unspecified) values, which any service provider should be able to agree to. Applications expect to be able to use this information about current network conditions to guide their use of the network, including any subsequent QOS requests. Note however, that information provided by the transport in the updated flow spec structure is only a hint and may be little more than a rough estimate that only applies to the first hop as opposed to the complete end-to-end connection. The application must take appropriate precautions in interpreting this information.

Connectionless sockets may also use **WSAConnect()** to establish a specified QOS level to a single designated peer. Otherwise connectionless sockets make use of **WSAIoctl()** to stipulate the initial QOS request, and any subsequent QOS renegotiations.

Even after a flow is established, conditions in the network may change or one of the communicating parties may invoke a QOS renegotiation which results in a reduction (or increase) in the available service level. A notification mechanism is included which utilizes the usual WinSock notification techniques (FD_QOS and FD_GROUP_QOS events) to indicate to the application that QOS levels have changed. The basic guideline for a service provider to generate FD_QOS/FD_GROUP_QOS notifications is when the current level of service supported is significantly different (especially in the negative direction) from what was last reported. The application should use **WSAIoctl()** with SIO_GET_QOS and/or SIO_GET_GROUP_QOS to retrieve the corresponding flow specs and examine them in order to discover what aspect of the service level has changed. Note that the QOS structures will be updated as appropriate regardless of whether FD_QOS/FD_GROUP_QOS is registered and generated. If the updated level of service is not acceptable, the application may adjust itself to accommodate it, attempt to renegotiate QOS, or close the socket. If a renegotiation is attempted, a successful return from the **WSAIoctl()** function indicates that the revised QOS request was accepted, otherwise an appropriate error will be indicated.

The flow specs proposed for WinSock 2 divide QOS characteristics into the following general areas:

1. Source Traffic Description - The manner in which the application's traffic will be injected into the network. This includes specifications for the token rate, the token bucket size, and the peak bandwidth. Note that even though the bandwidth requirement is expressed in terms of a token rate, this does not mean that service provider must actually implement token buckets. Any traffic management scheme that yields equivalent behavior is permitted.

2. Latency - Upper limits on the amount of delay and delay variation that are acceptable.

3. Level of service guarantee - Whether or not an absolute guarantee is required as opposed to best effort. Note that providers which have no feasible way to provide the level of service requested are expected to fail the connection attempt.

4. Provider-specific parameters - The flowspec itself can be extended in ways that are particular to specific providers.

2.8.1. The QOS Structure

The WinSock 2 QOS structure is defined through a combination of the qos.h and winsock2.h header files. The relevant definitions are summarized here.

```
typedef struct_WSABUF {
    u_long      len;     /* the length of the buffer */
    char FAR *  buf;     /* the pointer to the buffer */
} WSABUF, FAR * LPWSABUF;


typedef uint32   SERVICETYPE;
```

```
typedef struct _flowspec
{
    uint32      TokenRate;              /* In Bytes/sec */
    uint32      TokenBucketSize;        /* In Bytes */
    uint32      PeakBandwidth;          /* In Bytes/sec */
    uint32      Latency;                /* In microseconds */
    uint32      DelayVariation;         /* In microseconds */
    SERVICETYPE ServiceType;
    uint32      MaxSduSize;             /* In Bytes */
    uint32      MinimumPolicedSize;     /* In Bytes */

} FLOWSPEC, *PFLOWSPEC, FAR * LPFLOWSPEC;


typedef struct _QualityOfService
{
    FLOWSPEC      SendingFlowspec;    /* the flow spec for */
                                      /* data sending */
    FLOWSPEC      ReceivingFlowspec;  /* the flow spec for */
                                      /* data receiving */
    WSABUF        ProviderSpecific;   /* additional provider */
                                      /* specific stuff */
} QOS, FAR * LPQOS;
```

Definitions:

TokenRate/TokenBucketSize

A *Token bucket model* is used to specify the rate at which permission to send traffic (or credits) accrues. The value of -1 in these variables indicates that no rate limiting is in force. The *TokenRate* is expressed in bytes per second, and the *TokenBucketSize* in bytes.

The concept of the token bucket is a bucket which has a maximum volume (token bucket size) and continuously fills at a certain rate (token rate). If the “bucket” contains sufficient credit, the application may send data; if it does, it reduces the available credit by that amount. If sufficient credits are not available, the application must wait or discard the extra traffic.

If an application has been sending at a low rate for a period of time, it clearly may send a large burst of data all at once until it runs out of credit. Having done so, it must limit itself to sending at *TokenRate* until its data burst is exhausted.

In video applications, the *TokenRate* is typically the average bit rate peak to peak, and the *TokenBucketSize* is the largest typical frame size. In constant rate applications, the *TokenRate* is equal to the *PeakBandwidth*, and the *TokenBucketSize* is chosen to accommodate small variations.

PeakBandwidth

This field, expressed in bytes/second, limits how fast packets may be sent back to back from the application. Some intermediate systems can take advantage of this information resulting in a more efficient resource allocation.

Latency Latency is the maximum acceptable delay between transmission of a bit by the sender and its receipt by the intended receiver(s), expressed in microseconds. The precise interpretation of this number depends on the level of guarantee specified in the QOS request.

DelayVariation This field is the difference, in microseconds, between the maximum and minimum possible delay that a packet will experience. This value is used by applications to determine the amount of buffer space needed at the receiving side in order to restore the original data transmission pattern.

ServiceType This is the level of service being negotiated for. Values permitted for level of service are given below.

SERVICETYPE_NOTRAFFIC
In either Sending or Receiving flowspec, indicates that there will be no traffic in this direction. On duplex capable media, this signals underlying software to setup unidirectional connections only.

SERVICETYPE_BESTEFFORT
Indicates that the service provider, at minimum, takes the flow spec as a guideline and makes reasonable efforts to maintain the level of service requested, however without making any guarantees whatsoever.

SERVICETYPE_CONTROLLEDLOAD
Indicates that end-to-end behavior provided to an application by a series of network elements tightly approximates the behavior visible to applications receiving best-effort service "under unloaded conditions" from the same series of network elements. Thus, applications using this service may assume that: (1) A very high percentage of transmitted packets will be successfully delivered by the network to the receiving end-nodes. (Packet loss rate will closely approximate the basic packet error rate of the transmission medium).; and (2) Transit delay experienced by a very high percentage of the delivered packets will not greatly exceed the minimum transit delay experienced by any successfully delivered packet at the speed of light.

SERVICETYPE_GUARANTEED
Indicates that the service provider implements a queuing algorithm which isolates the flow from the effects of other flows as much as possible, and guarantees the flow the ability to propagate data at the *TokenRate* for the duration of the connection. If the sender sends faster than that rate, the network may delay or discard the excess traffic. If the sender does not exceed *TokenRate* over time, then latency is also guaranteed. This service type is designed for applications which require a precisely known quality of service but would

not benefit from better service, such as real-time control systems.

SERVICETYPE_NETWORK_UNAVAILABLE
In either a Sending or Receiving flowspec, this may be used by a service provider to indicate a loss of service in the corresponding direction.

SERVICETYPE_GENERAL_INFORMATION
Indicates that all service types are supported for this traffic flow.

SERVICETYPE_NOCHANGE
In either a Sending or Receiving flowspec, this requests that the QOS in the corresponding direction is not changed. This may be used when requesting a QOS change in one direction only, or when requesting a change only in the *ProviderSpecific* part of a QOS specification and not in the *SendingFlowspec* or the *ReceivingFlowspec*.

SERVICE_IMMEDIATE_TRAFFIC_CONTROL
In either a Sending or Receiving flowspec, this may be combined using bit-wise OR with one of the other defined *ServiceType* values to request the service provider to activate traffic control coincident with provision of the flowspec.

MaxSduSize The maximum packet size, in bytes, that is permitted or used in the traffic flow.

MinimumPolicedSize The minimum packet size that will be given the level of service requested.

2.8.2. QOS Templates

It is possible for QOS templates to be established for well-known media flows such as H.323, G.711, etc. The **WSAGetQOSByName()** function can be used to obtain the appropriate QOS structure for named media streams. It is up to each service provider to determine the appropriate values for each element in the QOS structure, as well as any protocol or media-dependent QOS extensions. The documentation for **WSAGetQOSByName()** will be periodically updated with a list of flow specifications and general descriptions as they become well-known. **WSAGetQOSByName()** can also be used to enumerate the set of known QOS template names.

2.8.3. Default Values

A default flow spec is associated with each eligible socket at the time it is created. Field values for this default flow spec are indicated below. In all cases these values indicate that no particular flow characteristics are being requested from the network. Applications only need to modify values for those fields which they are interested in, but must be aware that there exists some coupling between fields such as TokenRate and TokenBucketSize.

TokenRate =	0xFFFFFFFF (not specified)
TokenBucketSize =	0xFFFFFFFF (not specified)
PeakBandwidth =	0xFFFFFFFF (not specified)
Latency =	0xFFFFFFFF (not specified)

DelayVariation = 0xFFFFFFFF (not specified)
ServiceType = SERVICETYPE_NOCHANGE
MaxSduSize = 0xFFFFFFFF (not specified)
MinimumPolicedSize = 0xFFFFFFFF (not specified)

2.9. Socket Groups

Reserved for future use with socket groups:

> WinSock 2 introduces a number of function parameters, data types, structure members, and manifest constant values that are reserved for future use in grouping sockets together. As of the version 2.2.1 of the specification, the intended future use of these items is fully described, however, none of the group-related parameters is interpreted in software releases corresponding to the version 2.2.1 specification. Since a client always has the option to elect not to use socket groups, there are always default values and behaviors defined for group-related definitions. It is simple for an application that does not wish to use socket groups to use default values in such a fashion that the application will not be harmed if and when socket groups are "turned on" in the future. Definitions related to socket groups are marked in version 2.2.1 specification with the phrase: "Reserved for future use with socket groups" preceding the description of the intended future use.

WinSock 2 introduces the notion of a socket group as a means for an application (or cooperating set of applications) to indicate to an underlying service provider that a particular set of sockets are related and that the group thus formed has certain attributes. Group attributes include relative priorities of the individual sockets within the group and a group quality of service specification.

Applications needing to exchange multimedia streams over the network are benefited by being able to establish a specific relationship among the set of sockets being utilized. As a minimum this might include a hint to the service provider about the relative priorities of the media streams being carried. For example, a conferencing application would want to have the socket used for carrying the audio stream be given higher priority than that of the socket used for the video stream. Furthermore, there are transport providers (e.g. digital telephony and ATM) which can utilize a group quality of service specification to determine the appropriate characteristics for the underlying call or circuit connection. The sockets within a group are then multiplexed in the usual manner over this call. By allowing the application to identify the sockets that make up a group and to specify the required group attributes, such service providers can operate with maximum effectiveness.

WSASocket() and **WSAAccept()** are two new functions used to explicitly create and/or join a socket group coincident with creating a new socket. Socket group IDs can be retrieved by using **getsockopt()** with option SO_GROUP_ID. Relative priority can be accessed by using **get**/**setsockopt()** with option SO_GROUP_PRIORITY.

2.10. Shared Sockets

WSADuplicateSocket() is introduced to enable socket sharing across processes. A source process calls **WSADuplicateSocket()** to obtain a special WSAPROTOCOL_INFO structure for a target process ID. It uses some interprocess communications (IPC) mechanism to pass the contents of this structure to a target process. The target process then uses the WSAPROTOCOL_INFO structure in a call to **WSPSocket()**. The socket descriptor returned by this function will be an additional socket descriptor to an underlying socket which thus becomes shared. Note however, that sockets may be shared amongst threads in a given process without using the **WSADuplicateSocket()** function, since a socket descriptor is valid in all of a process' threads.

The two (or more) descriptors that reference a shared socket may be used independently as far as I/O is concerned. However, the WinSock interface does not implement any type of access control, so it is up to the processes involved to coordinate their operations on a shared socket. A typical use for shared sockets is to have one process that is responsible for creating sockets and establishing connections, hand off sockets to other processes which are responsible for information exchange.

Since what is duplicated are the socket descriptors and not the underlying socket, all of the state associated with a socket is held in common across all the descriptors. For example a **setsockopt()** operation performed using one descriptor is subsequently visible using a **getsockopt()** from any or all descriptors. A process may call **closesocket()** on a duplicated socket and the descriptor will become deallocated. The underlying socket, however, will remain open until **closesocket()** is called with the last remaining descriptor.

Notification on shared sockets is subject to the usual constraints of **WSAAsyncSelect()** and **WSAEventSelect()**. Issuing either of these calls using any of the shared descriptors cancels any previous event registration for the socket, regardless of which descriptor was used to make that registration. Thus, for example, it would not be possible to have process A receive FD_READ events and process B receive FD_WRITE events. For situations when such tight coordination is required, it is suggested that developers use threads instead of separate processes.

2.11. Enhanced Functionality During Connection Setup and Teardown

WSAAccept () allows an application to obtain caller information such as caller ID, QOS, etc., before deciding whether or not to accept an incoming connection request. This is done via a callback to an application-supplied condition function.

User-to-user data specified via parameters in **WSAConnect()** and/or the condition function of **WSAAccept()** may be transferred to the peer during connection establishment, provided this feature is supported by the service provider.

At connection teardown time, it is also possible (for protocols that support this) to exchange user data between the endpoints. The end that initiates the teardown can call **WSASendDisconnect()** to indicate that no more data is to be sent and cause the connection teardown sequence to be initiated. For certain protocols, part of this teardown sequence is the delivery of disconnect data from the teardown initiator. After receiving notice that the remote end has initiated the teardown sequence (typically via the FD_CLOSE indication), the **WSARecvDisconnect()** function may be called to receive the disconnect data (if any).

To illustrate how disconnect data might be used, consider the following scenario. The client half of a client/server application is responsible for terminating a socket connection. Coincident with the termination it provides (via disconnect data) the total number of transactions it processed with the server. The server in turn responds back with the cumulative grand total of transactions that it has processed with all clients. The sequence of calls and indications might occur as follows:

Client Side	Server Side
(1) invoke **WSASendDisconnect()** to conclude session and supply transaction total	
	(2) get FD_CLOSE, **recv()** with a return value of zero, or WSAEDISCON error return from **WSARecv()** indicating graceful shutdown in progress
	(3) invoke **WSARecvDisconnect()** to get client's transaction total
	(4) Compute cumulative grand total of all transactions
	(5) invoke **WSASendDisconnect()** to transmit grand total
(6) receive FD_CLOSE indication	(5') invoke **closesocket()**
(7) invoke **WSARecvDisconnect()** to receive and store cumulative grand total of transactions	
(8) invoke **closesocket()**	

Note that step (5') must follow step (5), but has no timing relationship with step (6), (7), or (8).

2.12. Extended Byte Order Conversion Routines

WinSock 2 does not assume that the network byte order for all protocols is the same. Therefore a set of conversion routines are supplied for converting 16 and 32 bit quantities to and from network byte order. These routines take as an input parameter the socket handle, which has a WSAPROTOCOL_INFO structure associated with it. The *NetworkByteOrder* field in the WSAPROTOCOL_INFO structure specifies what the desired network byte order is (currently either "big-endian" or "little-endian")..

2.13. Support for Scatter/Gather I/O

The **WSASend(), WSASendTo(), WSARecv(),** and **WSARecvFrom()** routines all take an array of application buffers as input parameters and thus may be used for scatter/gather (or vectored) I/O. This can be very useful in instances where portions of each message being transmitted consist of one or more fixed length "header" components in addition to message body. Such header components need not be concatenated by the application into a single contiguous buffer prior to sending. Likewise on receiving, the header components can be automatically split off into separate buffers, leaving the message body "pure".

When receiving into multiple buffers, completion occurs as data arrives from the network, regardless of whether all of the supplied buffers are utilized.

2.14. Protocol-Independent Multicast and Multipoint

Just as WinSock 2 allows the basic data transport capabilities of numerous transport protocols to be accessed in a generic manner, it also provides a generic way to utilize multipoint and multicast capabilities of transports that implement these features. To simplify, the term multipoint is used hereafter to refer to both multicast and multipoint communications.

Current multipoint implementations (e.g. IP multicast, ST-II, T.120, ATM UNI, etc.) vary widely with respect to how nodes join a multipoint session, whether a particular node is designated as a central or root node, and whether data is exchanged between all nodes or only between a root node and the various leaf nodes. WinSock 2's WSAPROTOCOL_INFO struct is used to declare the various multipoint attributes of a protocol. By examining these attributes the programmer will know what conventions to follow with the applicable WinSock 2 functions to setup, utilize and teardown multipoint sessions.

The features of WinSock 2 that support multicast can be summarized as follows:

- Two attribute bits in the WSAPROTOCOL_INFO struct
- Four flags defined for the *dwFlags* parameter of **WSASocket()**
- One function, **WSAJoinLeaf()**, for adding leaf nodes into a multipoint session
- Two **WSAIoctl()** command codes for controlling multipoint loopback and establishing the scope for multicast transmissions. (The latter corresponds to the IP multicast time-to-live or TTL parameter.)

Note that the inclusion of these multipoint features in WinSock 2 does not preclude an application from using an existing protocol-dependent interface, such as the Deering socket options for IP multicast (as described in the TCP/IP section of the ***Windows Sockets 2 Protocol-Specific Annex***).

Please refer to *Appendix B. Multipoint and Multicast Semantics* for detailed information on how the various multipoint schemes are characterized and how the applicable features of WinSock 2 are utilized.

2.15. Summary of New Socket Options

The new socket options for Winsock 2 are summarized in the following table. More detailed information is provided in section 3 under **getsockopt()** and/or **setsockopt()**. There are other new protocol-specific socket options which can be found in the protocol-specific annex.

Value	Type	Meaning	Default	Note
SO_GROUP_ID	GROUP	Reserved for future use with socket groups: The identifier of the group to which this socket belongs.	NULL	get only
SO_GROUP_PRIORITY	int	Reserved for future use with socket groups: The relative priority for sockets that are part of a socket group.	0	
SO_MAX_MSG_SIZE	int	Maximum outbound (send) size of a message for message-oriented socket types. There is no provision for finding out the maximum inbound message size. Has no meaning for stream-oriented sockets.	Implementation dependent	get only
SO_PROTOCOL_INFO	struct WSAPROTOCOL_INFO	Description of protocol info for protocol that is bound to this socket.	protocol dependent	get only
PVD_CONFIG	char FAR *	An opaque data structure object containing configuration information of the service provider.	Implementation dependent	

2.16. Summary of New Socket Ioctl Opcodes

The new socket ioctl opcodes for Winsock 2 are summarized in the following table. More detailed information is provided in section 3 under **WSAIoctl().** Note that **WSAIoctl()** also supports all the ioctl opcodes specified in **ioctlsocket()**. There are other new protocol-specific ioctl opcodes which can be found in the protocol-specific annex.

Opcode	Input Type	Output Type	Meaning
SIO_ASSOCIATE_HANDLE	companion API dependent	<not used>	Associate the socket with the specified handle of a companion interface.
SIO_ENABLE_CIRCULAR_QUEUEING	<not used>	<not used>	Circular queuing is enabled.
SIO_FIND_ROUTE	struct sockaddr	<not used>	Request the route to the specified address to be discovered.
SIO_FLUSH	<not used>	<not used>	Discard current contents of the sending queue.
SIO_GET_BROADCAST_ADDRESS	<not used>	struct sockaddr	Retrieve the protocol-specific broadcast address to be used in **sendto()/WSASendTo()**
SIO_GET_QOS	<not used>	QOS	Retrieve current flow spec(s) for the socket.
SIO_GET_GROUP_QOS	<not used>	QOS	Reserved for future use with socket groups: Retrieve current group flow spec(s) for the group this socket belongs to.
SIO_MULTIPOINT_LOOPBACK	BOOL	<not used>	Control whether data sent in a multipoint session will also be received by the same socket on the local host.
SIO_MULTICAST_SCOPE	int	<not used>	Specify the scope over which multicast transmissions will occur.
SIO_SET_QOS	QOS	<not used>	Establish new flow spec(s) for the socket.
SIO_SET_GROUP_QOS	QOS	<not used>	Reserved for future use with socket groups: Establish new group flow spec(s) for the group this socket belongs to.
SIO_TRANSLATE_HANDLE	int	companion API dependent	Obtain a corresponding handle for socket *s* that is valid in the context of a companion interface.
SIO_ROUTING_INTERFACE_QUERY	SOCKADDR	SOCKADDR	Obtain the address of local interface which should be used to send to the specified address
SIO_ROUTING_INTERFACE_CHANGE	SOCKADDR	<not used>	Request notification of changes in information reported via SIO_ROUTING_INTERFACE_QUERY for the specified address
SIO_ADDRESS_LIST_QUERY	<not used>	SOCKET_ADDRESS_LIST	Obtain the list of addresses to which application can bind.
SIO_ADDRESS_LIST_CHANGE	<not used>	<not used>	Request notification of changes in information reported via SIO_ADDRESS_LIST_QUERY

2.17. Summary of New Functions

The new API functions for Winsock 2 are summarized in the following tables.

2.17.1. Generic Data Transport Functions

WSAAccept()*	An extended version of **accept()** which allows for conditional acceptance and socket grouping.
WSACloseEvent()	Destroys an event object.
WSAConnect()*	An extended version of **connect()** which allows for exchange of connect data and QOS specification.
WSACreateEvent()	Creates an event object.
WSADuplicateSocket()	Create a new socket descriptor for a shared socket.
WSAEnumNetworkEvents()	Discover occurrences of network events.
WSAEnumProtocols()	Retrieve information about each available protocol.
WSAEventSelect()	Associate network events with an event object.
WSAGetOverlappedResult()	Get completion status of overlapped operation.
WSAGetQOSByName()	Supply QOS parameters based on a well-known service name.
WSAHtonl()	Extended version of **htonl()**
WSAHtons()	Extended version of **htons()**
WSAIoctl()*	Overlapped-capable version of **ioctlsocket()**
WSAJoinLeaf()*	Join a leaf node into a multipoint session.
WSANtohl()	Extended version of **ntohl()**
WSANtohs()	Extended version of **ntohs()**
WSAProviderConfigChange()	Receive notifications of service providers being installed/removed.
WSARecv()*	An extended version of **recv()** which accommodates scatter/gather I/O, overlapped sockets and provides the *flags* parameter as IN OUT
WSARecvDisconnect()	Terminate reception on a socket, and retrieve the disconnect data if the socket is connection-oriented.
WSARecvFrom()*	An extended version of **recvfrom()** which accommodates scatter/gather I/O, overlapped sockets and provides the *flags* parameter as IN OUT
WSAResetEvent()	Resets an event object.
WSASend()*	An extended version of **send()** which accommodates scatter/gather I/O and overlapped sockets
WSASendDisconnect()	Initiate termination of a socket connection and optionally send disconnect data.
WSASendTo()*	An extended version of **sendto()** which accommodates scatter/gather I/O and overlapped sockets
WSASetEvent()	Sets an event object.
WSASocket()	An extended version of **socket()** which takes a WSAPROTOCOL_INFO struct as input and allows overlapped sockets to be created. Also allows socket groups to be formed.
WSAWaitForMultipleEvents()	Blocks on multiple event objects.

* = The routine can block if acting on a blocking socket.

2.17.2. Name Registration and Resolution Functions

WSAAddressToString()	Convert an address structure into a human-readable

	numeric string
WSAEnumNameSpaceProviders()	Retrieve the list of available Name Registration and Resolution service providers
WSAGetServiceClassInfo	Retrieves all of the class-specific information pertaining to a service class
WSAGetServiceClassNameByClassId()	Returns the name of the service associated with the given type
WSAInstallServiceClass()	Create a new new service class type and store its class-specific information
WSALookupServiceBegin()	Initiate a client query to retrieve name information as constrained by a WSAQUERYSET data structure
WSALookupServiceEnd()	Finish a client query started by **WSALookupServiceBegin()** and free resources associated with the query
WSALookupServiceNext()	Retrieve the next unit of name information from a client query initiated by **WSALookupServiceBegin()**
WSARemoveServiceClass()	Permanently removes a service class type
WSASetService()	Register or deregister a service instance within one or more name spaces
WSAStringToAddress()	Convert a human-readable numeric string to a socket address structure suitable for passing to Windows Sockets routines.

3. Windows Sockets Programming Considerations

This section provides programmers with important information on a number of topics. It is especially pertinent to those who are porting socket applications from UNIX-based environments or who are upgrading their WinSock 1.1 applications to WinSock 2.

3.1. Deviation from BSD Sockets

There are a few limited instances where Windows Sockets has had to divert from strict adherence to the Berkeley conventions, usually because of important differences between UNIX and Windows environments.

3.1.1. Socket Data Type

A new data type, SOCKET, has been defined. This is needed because a WinSock application cannot assume that socket descriptors are equivalent to file descriptors as they are in UNIX. Furthermore, in UNIX, all handles, including socket handles, are small, non-negative integers, and some applications make assumptions that this will be true. WinSock handles have no restrictions, other than that the value INVALID_SOCKET is not a valid socket. Socket handles may take any value in the range 0 to INVALID_SOCKET-1.

Because the SOCKET type is unsigned, compiling existing source code from, for example, a UNIX environment may lead to compiler warnings about signed/unsigned data type mismatches.

This means, for example, that checking for errors when routines such as **socket()** or **accept()** return should not be done by comparing the return value with -1, or seeing if the value is negative (both common, and legal, approaches in BSD). Instead, an application should use the manifest constant INVALID_SOCKET as defined in Winsock2.h. For example:

TYPICAL BSD STYLE:

```
s = socket(...);
if (s == -1)      /* or s < 0 */
      {...}
```

PREFERRED STYLE:

```
s = socket(...);
if (s == INVALID_SOCKET)
      {...}
```

3.1.2. select() and FD_*

Because a SOCKET is no longer represented by the UNIX-style "small non-negative integer", the implementation of the **select()** function was changed in WinSock. Each set of sockets is still represented by the fd_set type, but instead of being stored as a bitmask the set is implemented as an array of SOCKETs. To avoid potential problems, applications must adhere to the use of the FD_XXX macros to set, initialize, clear, and check the fd_set structures.

3.1.3. Error codes - errno, h_errno & WSAGetLastError()

Error codes set by WinSock are NOT made available via the errno variable. Additionally, for the **getXbyY()** class of functions, error codes are NOT made available via the h_errno variable. Instead, error codes are accessed by using the **WSAGetLastError()** function described in section 4.33. This function is implemented in WinSock 2 as an alias for the Win32 function **GetLastError()**, and is intended to provide a reliable way for a thread in a multi-threaded process to obtain per-thread error information.

For compatibility with BSD, an application may choose to include a line of the form:

```
#define errno WSAGetLastError()
```

This will allow networking code which was written to use the global errno to work correctly in a single-threaded environment. There are, obviously, some drawbacks. If a source file includes code which inspects errno for both socket and non-socket functions, this mechanism cannot be used. Furthermore, it is not possible for an application to assign a new value to errno. (In WinSock the function **WSASetLastError()** may be used for this purpose.)

TYPICAL BSD STYLE:

```
r = recv(...);
if (r == -1
    && errno == EWOULDBLOCK)
      {...}
```

PREFERRED STYLE:

```
r = recv(...);
if (r == -1       /* (but see below) */
    && WSAGetLastError() == EWOULDBLOCK)
      {...}
```

Although error constants consistent with 4.3 Berkeley Sockets are provided for compatibility purposes, applications should, where possible, use the "WSA" error code definitions. This is because error codes returned by certain WinSock routines fall into the standard range of error codes as defined by Microsoft C. Thus, a better version of the above source code fragment is:

```
r = recv(...);
if (r == -1       /* (but see below) */
    && WSAGetLastError() == WSAEWOULDBLOCK)
      {...}
```

Note that this specification defines a recommended set of error codes, and lists the possible errors which may be returned as a result of each function. It may be the case in some implementations that other WinSock error codes will be returned in addition to those listed, and **applications should be prepared to handle errors other than those enumerated under each function description.** However WinSock will not return any value which is not enumerated in the table of legal WinSock errors given in Appendix A.1.

3.1.4. Pointers

All pointers used by applications with WinSock should be FAR, although this is only relevant to 16-bit applications, and meaningless in a 32-bit operating system. To facilitate this, data type definitions such as LPHOSTENT are provided.

3.1.5. Renamed functions

In two cases it was necessary to rename functions which are used in Berkeley Sockets in order to avoid clashes with other Windows APIs.

3.1.5.1. close() and closesocket()

In Berkeley Sockets, sockets are represented by standard file descriptors, and so the **close()** function can be used to close sockets as well as regular files. While nothing in the WinSock prevents an implementation from using regular file handles to identify sockets, nothing requires it either. Therefore, sockets must be closed by using the **closesocket()** routine. Using the **close()** routine to close a socket is incorrect and the effects of doing so are undefined by this specification.

3.1.5.2. ioctl() and ioctlsocket()/WSAIoctl()

Various C language run-time systems use the **ioctl()** routine for purposes unrelated to WinSock. For this reason we have defined the routine **ioctlsocket()** and **WSAIoctl()** which is used to handle socket functions which in the Berkeley Software Distribution are performed using **ioctl()** and **fcntl()**.

3.1.6. Maximum number of sockets supported

The maximum number of sockets supported by a particular WinSock service provider is implementation specific. An application should make no assumptions about the availability of a certain number of sockets. This topic is addressed further in section 4.54. , **WSAStartup()**. However, independent of the number of sockets supported by a particular implementation is the issue of the maximum number of sockets which an application can actually make use of.

The maximum number of sockets which a WinSock application can make use of is determined at the application's compile time by the manifest constant FD_SETSIZE. This value is used in constructing the fd_set structures used in **select()** (see section 4.16.). The default value in **Winsock2.h** is 64. If an application is designed to be capable of working with more than 64 sockets, the implementor should define the manifest FD_SETSIZE in every source file before including **Winsock2.h**. One way of doing this may be to include the definition within the compiler options in the makefile, for example adding -DFD_SETSIZE=128 as an option to the compiler command line for Microsoft C. It must be emphasized that defining FD_SETSIZE as a particular value has no effect on the actual number of sockets provided by a WinSock service provider.

3.1.7. Include files

For ease of portability of existing Berkeley sockets based source code, a number of standard Berkeley include files are supported. However, these Berkeley header files merely include the **Winsock2.h** include file, and it is therefore sufficient (and recommended) that WinSock application source files simply include **Winsock2.h**.

3.1.8. Return values on function failure

The manifest constant SOCKET_ERROR is provided for checking function failure. Although use of this constant is not mandatory, it is recommended. The following example illustrates the use of the SOCKET_ERROR constant:

TYPICAL BSD STYLE:

```
r = recv(...);
if (r == -1     /* or r < 0 */
    && errno == EWOULDBLOCK)
      {...}
```

PREFERRED STYLE:

```
r = recv(...);
if (r == SOCKET_ERROR
    && WSAGetLastError() == WSAEWOULDBLOCK)
      {...}
```

3.1.9. Raw Sockets

The WinSock specification does not mandate that a WinSock service provider support raw sockets, that is, sockets of type SOCK_RAW. However, service providers are allowed and encouraged to supply raw socket support. A WinSock-compliant application that wishes to use raw sockets should attempt to open the socket with the **socket()** call (see section 4.21.), and if it fails either attempt to use another socket type or indicate the failure to the user.

3.2. Byte Ordering

Care must always be taken to account for any differences between the Intel Architecture byte ordering and that used on the wire by individual transport protocols. Any reference to addresses or port numbers passed to or from a WinSock routine must be in the network order for the protocol being utilized. In the case of IP, this includes the IP address and port fields of a **struct sockaddr_in** (but not the *sin_family* field).

Consider an application which normally contacts a server on the TCP port corresponding to the "time" service, but which provides a mechanism for the user to specify that an alternative port is to be used. The port number returned by **getservbyname()** is already in network order, which is the format required for constructing an address, so no translation is required. However if the user elects to use a different port, entered as an integer, the application must convert this from host to TCP/IP network order (using the **WSAHtons()** function) before using it to construct an address. Conversely, if the application wishes to display the number of the port within an address (returned via, e.g., **getpeername()**), the port number must be converted from network to host order (using **WSANtohs()**) before it can be displayed.

Since the Intel Architecture and Internet byte orders are different, the conversions described above are unavoidable. Application writers are cautioned that they should use the standard conversion functions provided as part of WinSock rather than writing their own conversion code, since future implementations of WinSock are likely to run on systems for which the host order is identical to the network byte order. Only applications which use the standard conversion functions are likely to be portable.

3.3. WinSock 1.1 Compatibility Issues

To provide smooth backwards compatibility, WinSock 2 continues to support all of the WinSock 1.1 semantics and function calls except for those dealing with psuedo blocking. Since WinSock 2 runs only in 32 bit preemptively scheduled environments such as Windows NT and Windows 95, there is no need to implement the psuedo blocking found in WinSock 1.1. This means that the WSAEINPROGRESS error code will never be indicated and that the following WinSock 1.1 functions are **not** available to WinSock 2 applications:

1. **WSACancelBlockingCall()**
2. **WSAIsBlocking()**
3. **WSASetBlockingHook()**
4. **WSAUnhookBlockingHook()**

WinSock 1.1 programs that are written to utilize psuedo blocking will continue to operate correctly since they link to either WINSOCK.DLL or WSOCK32.DLL, both of which continue to support the complete set of WinSock 1.1 functions. In order for these programs to become WinSock 2 applications, some amount of code modification must occur. In most cases, judicious use of threads to accommodate processing that was being accomplished via a blocking hook function will suffice.

3.3.1. Default state for a socket's overlapped attribute

When Microsoft introduced the 32 bit version of WinSock 1.1 with their WSOCK32.DLL, the default case for the **socket()** function was to create sockets with the overlapped attribute. In order to preserve backwards compatibility with currently deployed WSOCK32.DLL implementations, this will continue to be the case for WinSock 2 as well. That is, in WinSock 2, sockets created via the **socket()** function will have the overlapped attribute. However, in order to be more compatible with the rest of the Win32 API, sockets created via **WSASocket()** will, by default, **not** have the overlapped attribute. This attribute will only be applied if the WSA_FLAG_OVERLAPPED flag bit is set.

3.3.2. Winsock 1.1 Blocking routines & EINPROGRESS

One major issue in porting applications from a Berkeley sockets environment to a WinSock 1.1 environment involves "blocking"; that is, invoking a function which does not return until the associated operation is completed. The problem arises when the operation may take an arbitrarily long time to

complete: an obvious example is a **recv()** which may block until data has been received from the peer system. The default behavior within the Berkeley sockets model is for a socket to operate in a blocking mode unless the programmer explicitly requests that operations be treated as non-blocking. WinSock 1.1 environments could not assume preemptive scheduling. Therefore, with WinSock 1.1 it was strongly recommended that programmers use the nonblocking (asynchronous) operations if at all possible. Because this was not always possible, the psuedo blocking facilities described below were provided (NOTE: This is no longer a necessary recommendation with WinSock 2, since it runs on pre-emptive 32-bit operating systems, where deadlocks are not a problem).

Even on a blocking socket, some operations (e.g. **bind()**, **getsockopt()**, **getpeername()**) can be completed immediately. For such operations there is no difference between blocking and non-blocking operation. Other operations (e.g. **recv()**) may be completed immediately or may take an arbitrary time to complete, depending on various transport conditions. When applied to a blocking socket, these operations are referred to as blocking operations.

With a (16-bit) WinSock 1.1, a blocking operation which cannot be completed immediately is handled via psuedo blocking as follows. The service provider initiates the operation, and then enters a loop in which it dispatches any Windows messages (yielding the processor to another thread if necessary) and then checks for the completion of the WinSock function. If the function has completed, or if **WSACancelBlockingCall()** has been invoked, the blocking function completes with an appropriate result. Refer to **WSASetBlockingHook()** for a complete description of this mechanism, including pseudo code for the various functions.

A service provider must allow installation of a blocking hook function that does not process messages in order to avoid the possibility of reentrant messages while a blocking operation is outstanding. The simplest such blocking hook function would simply return FALSE. If a service provider depends on messages for internal operation it may execute PeekMessage(hMyWnd...) before executing the application blocking hook so it can get its messages without affecting the rest of the system.

In (16-bit) WinSock 1.1 environments, if a Windows message is received for a process for which a blocking operation is in progress, there is a risk that the application will attempt to issue another WinSock call. Because of the difficulty of managing this condition safely, the WinSock 1.1 specification did not support such application behavior. In WinSock 1.1, it was not permissible for an application to make multiple nested Windows Sockets function calls. Only one outstanding function call was allowed for a particular task. The only exceptions being two functions that were provided to assist the programmer in this situation. **WSAIsBlocking()** may be called at any time to determine whether or not a blocking WinSock 1.1 call is in progress. Similarly, **WSACancelBlockingCall()** may be called at any time to cancel an in-progress blocking call, if any. Any other nesting of functions in WinSock 1.1 will fail with the error WSAEINPROGRESS.

It should be emphasized that this restriction applies to both blocking and non-blocking operations, but only for WinSock 1.1. For WinSock 2 applications (i.e., those that negotiate version 2.0 or higher at the time of **WSAStartup()**) there is no restriction on the nesting of operations. Operations can become nested under some rare circumstances such as during a **WSAAccept()** conditional-acceptance callback, or if a service provider in turn invokes a WinSock 2 function.

Although this mechanism is sufficient for simple applications, it cannot support the complex message-dispatching requirements of more advanced applications (for example, those using the MDI model). For such applications, the WinSock 1.1 API included the function **WSASetBlockingHook()**, which allows the application to specify a special routine that would be called instead of the default message dispatch routine described above.

The WinSock provider calls the blocking hook only if all of the following are true: the routine is one which is defined as being able to block, the specified socket is a blocking socket, and the request cannot

be completed immediately. (A socket is set to blocking by default, but the IOCTL FIONBIO, and **WSAAsyncSelect()**set a socket to nonblocking mode.) If a WinSock 1.1 application uses only non-blocking sockets and uses the **WSAAsyncSelect()** and/or the **WSAAsyncGetXByY()** routines instead of **select()** and the **getXbyY()** routines, then the blocking hook will never be called and the application does not need to be concerned with the reentrancy issues the blocking hook can introduce.

If a WinSock 1.1 application invokes an asynchronous or non-blocking operation which takes a pointer to a memory object (e.g. a buffer, or a global variable) as an argument, it is the responsibility of the application to ensure that the object is available to WinSock throughout the operation. The application must not invoke any Windows function which might affect the mapping or addressability of the memory involved.

3.4. Graceful shutdown, linger options and socket closure

Much confusion has been evidenced on the subject of shutting down socket connections and closing the sockets involved. Unless properly understood, the recent addition of **WSASendDisconnect()** and **WSARecvDisconnect()** could possibly exacerbate this situation. The following material is provided as clarification.

It is important to distinguish the difference between shutting down a socket connection and closing a socket. Shutting down a socket connection involves an exchange of protocol messages between the two endpoints which is hereafter referred to as a shutdown sequence. Two general classes of shutdown sequences are defined: graceful and abortive (also referred to as “hard”). In a graceful shutdown sequence, any data that has been queued but not yet transmitted can be sent prior to the connection being closed. In an abortive shutdown, any unsent data is lost. The occurrence of a shutdown sequence (graceful or abortive) can also be used to provide an FD_CLOSE indication to the associated applications signifying that a shutdown is in progress. Closing a socket, on the other hand, causes the socket handle to become deallocated so that the application can no longer reference or use the socket in any manner.

In Windows Sockets, both the **shutdown()** function, and the **WSASendDisconnect()** function can be used to initiate a shutdown sequence, while the **closesocket()** function is used to deallocate socket handles and free up any associated resources. Some amount of confusion arises, however, from the fact that the **closesocket()** function will implicitly cause a shutdown sequence to occur if it has not already happened. In fact, it has become a rather common programming practice to rely on this feature and use **closesocket()** to both initiate the shutdown sequence and deallocate the socket handle.

To facilitate this usage, the sockets interface provides for controls via the socket option mechanism that allows the programmer to indicate whether the implicit shutdown sequence should be graceful or abortive, and also whether the **closesocket()** function should linger (i.e. not complete immediately) to allow time for a graceful shutdown sequence to complete. Provided that the programmer fully understands the ramifications of using **closesocket()** in this manner, all is well. Unfortunately, many do not.

By establishing appropriate values for the socket options SO_LINGER and SO_DONTLINGER, the following types of behavior can be obtained with the **closesocket()** function.

- Abortive shutdown sequence, immediate return from **closesocket().**

- Graceful shutdown, delay return until either shutdown sequence completes or a specified time interval elapses. If the time interval expires before the graceful shutdown sequence completes, an abortive shutdown sequence occurs and **closesocket()** returns.

- Graceful shutdown, return immediately and allow the shutdown sequence to complete in the background. This is the default behavior. Note, however, that the application has no way of knowing when (or whether) the graceful shutdown sequence completes.

One technique that can be used to minimize the chance of problems occurring during connection teardown is to not rely on an implicit shutdown being initiated by **closesocket()**. Instead one of the two explicit shutdown functions (**shutdown()** or **WSASendDisconnect()**) are used. This in turn will cause an FD_CLOSE indication to be received by the peer application indicating that all pending data has been received. To illustrate this, the following table shows the functions that would be invoked by the client and server components of an application, where the client is responsible for initiating a graceful shutdown.

Client Side	Server Side
(1) Invoke **shutdown(*s, SD_SEND)*** to signal end of session and that client has no more data to send.	
	(2) Receive FD_CLOSE, indicating graceful shutdown in progress and that all data has been received.
	(3) Send any remaining response data.
(5') Get FD_READ and invoke **recv()** to get any response data sent by server	(4) Invoke **shutdown(*s, SD_SEND*)** to indicate server has no more data to send.
(5) Receive FD_CLOSE indication	(4') Invoke **closesocket()**
(6) Invoke **closesocket()**	

Note that the timing sequence is maintained from step (1) to step (6) between the client and the server, except for step (4') and (5') which only has local timing significance in the sense that step (5) follows step (5') on the client side while step (4') follows step (4) on the server side, with no timing relationship with the remote party.

3.5. Out-Of-Band data

The stream socket abstraction includes the notion of "out of band" (OOB) data. Many protocols allow portions of incoming data to be marked as "special" in some way, and these special data blocks may be delivered to the user out of the normal sequence - examples include "expedited data" in X.25 and other OSI protocols, and BSD Unix's use of "urgent data" in TCP. This section describes OOB data handling in a protocol-independent manner - a discussion of OOB data implemented using TCP "urgent data" follows. Note that in the following, the use of **recv()** also implies **recvfrom(), WSARecv(), and WSARecvFrom()**, and references to **WSAAsyncSelect()** also apply to **WSAEventSelect()**.

3.5.1. Protocol Independent OOB data

OOB data is a logically independent transmission channel associated with each pair of connected stream sockets. OOB data may be delivered to the user independently of normal data. The abstraction defines that the OOB data facilities must support the reliable delivery of at least one OOB data block at a time. This data block may contain at least one byte of data, and at least one OOB data block may be pending delivery to the user at any one time. For communications protocols which support in-band signaling (i.e. TCP, where the "urgent data" is delivered in sequence with the normal data), the system normally extracts the OOB data from the normal data stream and stores it separately (leaving a gap in the "normal" data stream). This allows users to choose between receiving the OOB data in order and receiving it out of sequence without having to buffer all the intervening data. It is possible to "peek" at out-of-band data.

A user can determine if there is any OOB data waiting to be read using the **ioctlsocket**(SIOCATMARK) function (q.v.). For protocols where the concept of the "position" of the OOB data block within the normal data stream is meaningful (i.e. TCP), a Windows Sockets service provider will maintain a conceptual "marker" indicating the position of the last byte of OOB data within the normal data stream.

This is not necessary for the implementation of the **ioctlsocket**(SIOCATMARK) functionality - the presence or absence of OOB data is all that is required.

For protocols where the concept of the "position" of the OOB data block within the normal data stream is meaningful an application may prefer to process out-of-band data "in-line", as part of the normal data stream. This is achieved by setting the socket option SO_OOBINLINE (see section 4.19. , **setsockopt()**). For other protocols where the OOB data blocks are truly independent of the normal data stream, attempting to set SO_OOBINLINE will result in an error. An application can use the SIOCATMARK **ioctlsocket()** command (see section 4.10.) to determine whether there is any unread OOB data preceding the mark. For example, it might use this to resynchronize with its peer by ensuring that all data up to the mark in the data stream is discarded when appropriate.

With SO_OOBINLINE disabled (the default setting):

- WinSock notifies an application of an FD_OOB event, if the application registered for notification with **WSAAsyncSelect()**, in exactly the same way FD_READ is used to notify of the presence of normal data. That is, FD_OOB is posted when OOB data arrives and there was no OOB data previously queued, and also when data is read using the MSG_OOB flag, and some OOB data remains to be read after the read operation has returned. FD_READ messages are not posted for OOB data.
- WinSock returns from **select()** with the appropriate *exceptfds* socket set if OOB data is queued on the socket.
- the application can call **recv()** with MSG_OOB to read the urgent data block at any time. The block of OOB data "jumps the queue".
- the application can call **recv()** without MSG_OOB to read the normal data stream. The OOB data block will not appear in the data stream with "normal data." If OOB data remains after any call to **recv()**, WinSock notifies the application with FD_OOB or via *exceptfds* when using **select()**.
- For protocols where the OOB data has a position within the normal data stream, a single **recv()** operation will not span that position. One **recv()** will return the normal data before the "mark", and a second **recv()** is required to begin reading data after the "mark".

With SO_OOBINLINE enabled:

- FD_OOB messages are _NOT_ posted for OOB data - for the purpose of the **select()** and **WSAAsyncSelect()** functions, OOB data is treated as normal data, and indicated by setting the socket in *readfds* or by sending an FD_READ message respectively.
- the application may not call **recv()** with the MSG_OOB flag set to read the OOB data block - the error code WSAEINVAL will be returned.
- the application can call **recv()** without the MSG_OOB flag set. Any OOB data will be delivered in its correct order within the "normal" data stream. OOB data will never be mixed with normal data - there must be three read requests to get past the OOB data. The first returns the normal data prior to the OOB data block, the second returns the OOB data, the third returns the normal data following the OOB data. In other words, the OOB data block boundaries are preserved.

The **WSAAsyncSelect()** routine is particularly well suited to handling notification of the presence of out-of-band-data when SO_OOBINLINE is off.

3.5.2. OOB data in TCP

Note: The following discussion of out-of-band (OOB) data, implemented using TCP Urgent data, follows the model used in the Berkeley software distribution. Users and implementors should be aware of the fact that there are at present two conflicting interpretations of RFC 793 (in which the concept is introduced), and that the implementation of out-of-band data in the Berkeley Software Distribution (BSD) does not conform to the Host Requirements laid down in RFC 1122.

Specifically, the TCP urgent pointer in BSD points to the byte after the urgent data byte, and an RFC-compliant TCP urgent pointer points to the urgent data byte. As a result, if an application sends urgent data from a BSD-compatible implementation to an RFC-1122 compatible implementation then the receiver will read the wrong urgent data byte (it will read the byte located after the correct byte in the data stream as the urgent data byte).

To minimize interoperability problems, applications writers are advised not to use out-of-band data unless this is required in order to interoperate with an existing service. Windows Sockets suppliers are urged to document the out-of-band semantics (BSD or RFC 1122) which their product implements.

Arrival of a TCP segment with the "URG"ent flag set indicates the existence of a single byte of "OOB" data within the TCP data stream. The "OOB data block" is one byte in size. The urgent pointer is a positive offset from the current sequence number in the TCP header that indicates the location of the "OOB data block" (ambiguously, as noted above). This may point to data that has not yet been received.

With SO_OOBINLINE disabled (the default), when the TCP segment containing the byte pointed to by the urgent pointer arrives, the OOB data block (one byte) is removed from the data stream and buffered. If a subsequent TCP segment arrives with the urgent flag set (and a new urgent pointer), the OOB byte currently queued may be lost as it is replaced by the new OOB data block (as occurs in Berkeley Software Distribution). It is never replaced in the data stream, however.

With SO_OOBINLINE enabled, the urgent data remains in the data stream. As a result, the OOB data block is never lost when a new TCP segment arrives containing urgent data. The existing OOB data "mark" is updated to the new position.

3.6. Summary of WinSock 2 Functions

The following tables summarize the functions included in WinSock 2. (NOTE: It does not include the ***Name Resolution and Registration*** functions described in section 4.)

3.6.1. BSD Socket Functions

The WinSock specification includes the following Berkeley-style socket routines (NOTE: All of the following functions were part of the WinSock 1.1 API):

accept() *	An incoming connection is acknowledged and associated with an immediately created socket. The original socket is returned to the listening state.
bind()	Assign a local name to an unnamed socket.
closesocket() *	Remove a socket from the per-process object reference table. Only blocks if SO_LINGER is set with a non-zero timeout on a blocking socket.
connect() *	Initiate a connection on the specified socket.
getpeername()	Retrieve the name of the peer connected to the specified socket.
getsockname()	Retrieve the local address to which the specified socket is bound.
getsockopt()	Retrieve options associated with the specified socket.
htonl() ∞	Convert a 32-bit quantity from host byte order to network byte order.
htons() ∞	Convert a 16-bit quantity from host byte order to network byte order.
inet_addr() ∞	Converts a character string representing a number in the Internet standard "." notation to an Internet address value.
inet_ntoa() ∞	Converts an Internet address value to an ASCII string in "." notation i.e. "a.b.c.d".
ioctlsocket()	Provide control for sockets.
listen()	Listen for incoming connections on a specified socket.
ntohl() ∞	Convert a 32-bit quantity from network byte order to host byte order.
ntohs() ∞	Convert a 16-bit quantity from network byte order to host byte order.
recv() *	Receive data from a connected or unconnected socket.
recvfrom() *	Receive data from either a connected or unconnected socket.
select() *	Perform synchronous I/O multiplexing.
send() *	Send data to a connected socket.
sendto() *	Send data to either a connected or unconnected socket.
setsockopt()	Store options associated with the specified socket.
shutdown()	Shut down part of a full-duplex connection.
socket()	Create an endpoint for communication and return a socket descriptor.

* = The routine can block if acting on a blocking socket.
∞ = The routine is retained for backward compatibility with WinSock 1.1, and should only be used for sockets created with AF_INET address family.

3.6.2. Microsoft Windows-specific Extension Functions

The WinSock specification provides a number of extensions to the standard set of Berkeley Sockets routines. Principally, these extended functions allow message or function-based, asynchronous access to network events, as well as enable overlapped I/O. While use of this extended API set is not mandatory for socket-based programming (with the exception of **WSAStartup()** and **WSACleanup()**), it is recommended for conformance with the Microsoft Windows programming paradigm. For features introduced in WinSock 2, please see section 2 for details.

WSAAccept()*	An extended version of **accept()** which allows for conditional acceptance and socket grouping.
WSAAsyncGetHostByAddr()∞****	A set of functions which provide asynchronous versions of the standard Berkeley **getXbyY()** functions. For example, the **WSAAsyncGetHostByName()** function provides an asynchronous message based implementation of the standard Berkeley **gethostbyname()** function.
WSAAsyncGetHostByName()∞****	
WSAAsyncGetProtoByName()∞****	
WSAAsyncGetProtoByNumber()∞***	
WSAAsyncGetServByName()∞****	
WSAAsyncGetServByPort()∞****	
WSAAsyncSelect()**	Perform asynchronous version of **select()**
WSACancelAsyncRequest()∞****	Cancel an outstanding instance of a **WSAAsyncGetXByY()** function.
WSACleanup()	Sign off from the underlying WinSock DLL.
WSACloseEvent()	Destroys an event object.
WSAConnect()*	An extended version of **connect()** which allows for exchange of connect data and QOS specification.
WSACreateEvent()	Creates an event object.
WSADuplicateSocket()	Allow an underlying socket to be shared by creating a virtual socket.
WSAEnumNetworkEvents()	Discover occurrences of network events.
WSAEnumProtocols()	Retrieve information about each available protocol.
WSAEventSelect()	Associate network events with an event object.
WSAGetLastError()**	Obtain details of last WinSock error
WSAGetOverlappedResult()	Get completion status of overlapped operation.
WSAGetQOSByName()	Supply QOS parameters based on a well-known service name.
WSAHtonl()	Extended version of **htonl()**
WSAHtons()	Extended version of **htons()**
WSAIoctl()*	Overlapped-capable version of **ioctl()**
WSAJoinLeaf()*	Add a multipoint leaf to a multipoint session
WSANtohl()	Extended version of **ntohl()**
WSANtohs()	Extended version of **ntohs()**
WSAProviderConfigChange()	Receive notifications of service providers being installed/removed.
WSARecv()*	An extended version of **recv()** which accommodates scatter/gather I/O, overlapped sockets and provides the *flags* parameter as IN OUT
WSARecvFrom()*	An extended version of **recvfrom()** which accommodates scatter/gather I/O, overlapped sockets and provides the *flags* parameter as IN OUT
WSAResetEvent()	Resets an event object.
WSASend()*	An extended version of **send()** which accommodates scatter/gather I/O and overlapped sockets
WSASendTo()*	An extended version of **sendto()** which accommodates scatter/gather I/O and overlapped sockets
WSASetEvent()	Sets an event object.
WSASetLastError()**	Set the error to be returned by a subsequent **WSAGetLastError()**

WSASocket()	An extended version of **socket()** which takes a WSAPROTOCOL_INFO struct as input and allows overlapped sockets to be created. Also allows socket groups to be formed.
WSAStartup()**	Initialize the underlying WinSock DLL.
WSAWaitForMultipleEvents()*	Blocks on multiple event objects.

* = The routine can block if acting on a blocking socket.
∞ = The routine is realized by queries to name space providers that supports AF_INET, if any
** = The routine was originally a WinSock 1.1 function.

4. SOCKET LIBRARY REFERENCE

This chapter presents the data transport routines in alphabetical order, and describes each routine in detail.

Note: the getXbyY() and WSAAsyncGetXbyY() functions now appear in section 5 on name resolution.

In each routine it is indicated that the header file **winsock2.h** must be included. Appendix A.2 lists the Berkeley-compatible header files which are supported. These are provided for compatibility purposes only, and each of them will simply include **winsock2.h**. The Windows header file **windows.h** is also needed, but **winsock2.h** will include it if necessary.

4.1. accept()

Description Accept a connection on a socket.

```
#include <winsock2.h>

SOCKET WSAAPI
accept (
        IN      SOCKET                  s,
        OUT     struct sockaddr FAR*    addr,
        OUT     int  FAR*               addrlen
);
```

s — A descriptor identifying a socket which is listening for connections after a **listen()**.

addr — An optional pointer to a buffer which receives the address of the connecting entity, as known to the communications layer. The exact format of the *addr* argument is determined by the address family established when the socket was created.

addrlen — An optional pointer to an integer which contains the length of the address *addr*.

Remarks This routine extracts the first connection on the queue of pending connections on *s*, creates a new socket and returns a handle to the new socket. The newly created socket has the same properties as *s* including asynchronous events registered with **WSAAsyncSelect()** or with **WSAEventSelect()**, but **not** including the listening socket's group ID, if any. If no pending connections are present on the queue, and the socket is not marked as non-blocking, **accept()** blocks the caller until a connection is present.

If the socket is marked non-blocking and no pending connections are present on the queue, **accept()** returns a failure with the error WSAEWOULDBLOCK, as described below.

After accept succeeds, and returns a new socket handle, the accepted socket may not be used to accept more connections. The original socket remains open, and listening for new connection requests.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the address family in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-oriented socket types such as SOCK_STREAM. If *addr* and/or *addrlen* are equal to NULL, then no information about the remote address of the accepted socket is returned.

Return Value

If no error occurs, **accept()** returns a value of type SOCKET which is a descriptor for the accepted socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

The integer referred to by *addrlen* initially contains the amount of space pointed to by *addr*. On return it will contain the actual length in bytes of the address returned.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *addrlen* argument is too small or *addr* is not a valid part of the user address space.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	**listen()** was not invoked prior to **accept()**.
WSAEMFILE	The queue is non-empty upon entry to **accept()** and there are no descriptors available.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The referenced socket is not a type that supports connection-oriented service.
WSAEWOULDBLOCK	The socket is marked as non-blocking and no connections are present to be accepted.

See Also

bind(), **connect()**, **listen()**, **select()**, **socket()**, **WSAAsyncSelect(), WSAAccept()**

4.2. bind()

Description Associate a local address with a socket.

```
#include <winsock2.h>

int WSAAPI
bind (
        IN      SOCKET                          s,
        IN      const struct sockaddr FAR*      name,
        IN      int                             namelen
);
```

s A descriptor identifying an unbound socket.

name The address to assign to the socket. The sockaddr structure is defined as follows:

```
struct sockaddr {
      u_short     sa_family;
      char        sa_data[14];
};
```

Except for the `sa_family` field, `sockaddr` contents are expressed in network byte order. NOTE: In WinSock 2, the *name* parameter is not strictly interpreted as a pointer to a "sockaddr" struct. It is cast this way for Windows Sockets compatibility. The actual structure is interpreted differently in the context of different address families. The only requirements are that the first u_short is the address family and the total size of the memory buffer in bytes is *namelen*

namelen The length of the *name*.

Remarks This routine is used on an unconnected connectionless or connection-oriented socket, before subsequent **connect()**s or **listen()**s. When a socket is created with **socket()**, it exists in a name space (address family), but it has no name assigned. **bind()** establishes the local association of the socket by assigning a local name to an unnamed socket.

As an example, in the Internet address family, a name consists of three parts: the address family, a host address, and a port number which identifies the application. In WinSock 2, the *name* parameter is not strictly interpreted as a pointer to a "sockaddr" struct. It is cast this way for Windows Sockets compatibility. Service Providers are free to regard it as a pointer to a block of memory of size *namelen*. The first two bytes in this block (corresponding to "sa_family" in the "sockaddr" declaration) must contain the address family that was used to create the socket. Otherwise an error WSAEFAULT will occur.

If an application does not care what local address is assigned to it, it may specify the manifest constant value ADDR_ANY for the sa_data field of the name parameter. This allows the underlying service provider to use any appropriate network address, potentially simplifying application programming in the presence of multi-homed hosts (i.e., hosts that have more than one network interface and address). For TCP/IP, if the port is specified as 0, the service provider will assign a unique port to the application with a value between 1024 and 5000. The application may use **getsockname()** after

bind() to learn the address and the port that has been assigned to it, but note that if the Internet address is equal to INADDR_ANY, **getsockname()** will not necessarily be able to supply the address until the socket is connected, since several addresses may be valid if the host is multi-homed. Binding to a specific port number (i.e., other than port 0) is discouraged for client applications, since there is a danger of conflicting with another socket that is already using that port number.

Return Value

If no error occurs, **bind()** returns 0. Otherwise, it returns SOCKET_ERROR, and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	Some process on the machine has already bound to the same fully-qualified address (e.g., IP address and port in the af_inet case) and the socket has not been marked to allow address re-use with SO_REUSEADDR. (See the SO_REUSEADDR socket option under **setsockopt()**.)
WSAEADDRNOTAVAIL	The specified address is not a valid address for this machine.
WSAEFAULT	The *name* or the *namelen* argument is not a valid part of the user address space, the *namelen* argument is too small, the *name* argument contains incorrect address format for the associated address family, or the first two bytes of the memory block specified by *name* does not match the address family associated with the socket descriptor *s*.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The socket is already bound to an address.
WSAENOBUFS	Not enough buffers available, too many connections.
WSAENOTSOCK	The descriptor is not a socket.

See Also

connect(), **listen()**, **getsockname()**, **setsockopt()**, **socket()**, **WSACancelBlockingCall()**.

4.3. closesocket()

Description Close a socket.

```
#include <winsock2.h>

int WSAAPI
closesocket (
        IN      SOCKET      s
);
```

s A descriptor identifying a socket.

Remarks This function closes a socket. More precisely, it releases the socket descriptor *s*, so that further references to *s* will fail with the error WSAENOTSOCK. If this is the last reference to an underlying socket, the associated naming information and queued data are discarded. Any pending blocking or asynchronous calls issued by any thread in this process are canceled without posting any notification messages. Any pending overlapped operations (e.g., **WSASend()**/**WSASendTo()**/**WSARecv()**/**WSARecvFrom()**/**WSAIoctl()** with an overlapped socket) issued by any thread in this process are also canceled. Whatever completion action was specified for these overlapped operations is performed (e.g., event, completion routine, or completion port). In this case, the pending overlapped operations fail with the error status WSA_OPERATION_ABORTED. An application should always have a matching call to **closesocket()** for each successful call to **socket()** to return socket resources to the system.

The semantics of **closesocket()** are affected by the socket options SO_LINGER and SO_DONTLINGER as follows (Note: by default SO_DONTLINGER is enabled - that is, SO_LINGER is disabled):

Option	Interval	Type of close	Wait for close?
SO_DONTLINGER	Don't care	Graceful	No
SO_LINGER	Zero	Hard	No
SO_LINGER	Non-zero	Graceful	Yes

If SO_LINGER is set (i.e. the *l_onoff* field of the linger structure is non-zero; see sections 4.7. and 4.19.) with a zero timeout interval (*l_linger* is zero), **closesocket()** is not blocked even if queued data has not yet been sent or acknowledged. This is called a "hard" or "abortive" close, because the socket's virtual circuit is reset immediately, and any unsent data is lost. Any **recv()** call on the remote side of the circuit will fail with WSAECONNRESET.

If SO_LINGER is set with a non-zero timeout interval on a blocking socket, the **closesocket()** call blocks on a blocking socket until the remaining data has been sent or until the timeout expires. This is called a graceful disconnect. If the timeout expires before all data has been sent, the Windows Sockets implementation aborts the connection before **closesocket()** returns.

Enabling SO_LINGER with a non-zero timeout interval on a non-blocking socket is not recommended. In this case, the call to **closesocket()** will fail with an error of WSAEWOULDBLOCK if the close operation cannot be completed immediately. If **closesocket()** fails with WSAEWOULDBLOCK the socket handle is still valid, and a

disconnect is not initiated. The application must call **closesocket()** again to close the socket, although **closesocket()** may continue to fail unless the application disables SO_DONTLINGER, enables SO_LINGER with a zero timeout, or calls **shutdown()** to initiate closure.

If SO_DONTLINGER is set on a stream socket (i.e. the *l_onoff* field of the linger structure is zero; see sections 4.7. and 4.19.), the **closesocket()** call will return immediately and does not get WSAEWOULDBLOCK, whether the socket is blocking or non-blocking. However, any data queued for transmission will be sent if possible before the underlying socket is closed. This is also called a graceful disconnect. Note that in this case the WinSock provider may not release the socket and other resources for an arbitrary period, which may affect applications which expect to use all available sockets. This is the default behavior (SO_DONTLINGER is set by default).

Note: to assure that all data is sent and received on a connection, an application should call **shutdown()** before calling **closesocket()** (see section *3.4. Graceful shutdown, linger options and socket closure* for more information). Also note, FD_CLOSE will not be posted after **closesocket()** is called.

Here is a summary of **closesocket()** behavior:

- if SO_DONTLINGER enabled (the default setting) it always returns immediately without WSAEWOULDBLOCK - connection is gracefully closed "in the background"
- if SO_LINGER enabled with a zero timeout: it always returns immediately - connection is reset/aborted
- if SO_LINGER enabled with non-zero timeout:
  - with blocking socket it blocks until all data sent or timeout expires
  - with non-blocking socket it returns immediately indicating failure

For additional information please see *3.4. Graceful shutdown, linger options and socket closure*.

Return Value

If no error occurs, **closesocket()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**
WSAEWOULDBLOCK	The socket is marked as nonblocking and SO_LINGER is set to a nonzero timeout value.

See Also **accept()**, **socket()**, **ioctlsocket()**, **setsockopt()**, **WSAAsyncSelect()**, **WSADuplicateSocket()**.

4.4. connect()

Description Establish a connection to a peer.

```
#include <winsock2.h>

int WSAAPI
connect (
        IN      SOCKET                          s,
        IN      const struct sockaddr FAR*      name,
        IN      int                             namelen
);
```

s A descriptor identifying an unconnected socket.

name The name of the peer to which the socket is to be connected.

namelen The length of the *name*.

Remarks This function is used to create a connection to the specified destination. If the socket, *s*, is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound.

For connection-oriented sockets (e.g., type SOCK_STREAM), an active connection is initiated to the foreign host using *name* (an address in the name space of the socket; for a detailed description, please see **bind()**). When the socket call completes successfully, the socket is ready to send/receive data. If the address field of the *name* structure is all zeroes, **connect()** will return the error WSAEADDRNOTAVAIL. Any attempt to re-connect an active connection will fail with the error code WSAEISCONN.

For connection-oriented, non-blocking sockets it is often not possible to complete the connection immediately. In such a case, this function returns with the error WSAEWOULDBLOCK but the operation proceeds. When the success or failure outcome becomes known, it may be reported in one of several ways depending on how the client registers for notification. If the client uses **select()** success is reported in the **writefds** set and failure is reported in the **exceptfds** set. If the client uses **WSAAsyncSelect()** or **WSAEventSelect()**, the notification is announced with FD_CONNECT and the error code associated with the FD_CONNECT indicates either success or a specific reason for failure.

For a connectionless socket (e.g., type SOCK_DGRAM), the operation performed by **connect()** is merely to establish a default destination address which will be used on subsequent **send()**/**WSASend()** and **recv()**/**WSARecv()** calls. Any datagrams received from an address other than the destination address specified will be discarded. If the address field of the *name* structure is all zeroes, the socket will be "dis-connected" - the default remote address will be indeterminate, so **send()/WSASend()** and **recv()/WSARecv()** calls will return the error code WSAENOTCONN, although **sendto()/WSASendTo()** and **recvfrom()/WSARecvFrom()** may still be used. The default destination may be changed by simply calling **connect()** again, even if the socket is already "connected". Any datagrams queued for receipt are discarded if *name* is different from the previous **connect()**.

For connectionless sockets, *name* may indicate any valid address, including a broadcast address. However, to connect to a broadcast address, a socket must have **setsockopt()** SO_BROADCAST enabled, otherwise **connect()** will fail with the error code WSAEACCES.

Comments

When connected sockets break (i.e. become closed for whatever reason), they should be discarded and recreated. It is safest to assume that when things go awry for any reason on a connected socket, the application must discard and recreate the needed sockets in order to return to a stable point.

Return Value

If no error occurs, **connect()** returns 0. Otherwise, it returns SOCKET_ERROR, and a specific error code may be retrieved by calling **WSAGetLastError().**

On a blocking socket, the return value indicates success or failure of the connection attempt.

With a non-blocking socket, the connection attempt may not be completed immediately - in this case **connect()** will return SOCKET_ERROR, and **WSAGetLastError()** will return WSAEWOULDBLOCK. In this case the application may:

1. Use **select()** to determine the completion of the connection request by checking if the socket is writeable, or

2. If your application is using **WSAAsyncSelect()** to indicate interest in connection events, then your application will receive an FD_CONNECT notification when the connect operation is complete (successfully, or not), or

3. If your application is using **WSAEventSelect()** to indicate interest in connection events, then the associated event object will be signaled when the connect operation is complete (successfully, or not).

For a non-blocking socket, until the connection attempt completes, all subsequent calls to **connect()** on the same socket will fail with the error code WSAEALREADY, and WSAEISCONN when the connection completes successfully. Due to ambiguities in version 1.1 of the Windows Sockets specification, error codes returned from **connect()** while a connection is already pending may vary among implementations. As a result, it isn't recommended that applications use multiple calls to **connect()** to detect connection completion. If they do, they must be prepared to handle WSAEINVAL and WSAEWOULDBLOCK error values the same way that they handle WSAEALREADY, to assure robust execution.

If the return error code indicates the connection attempt failed (i.e. WSAECONNREFUSED, WSAENETUNREACH, WSAETIMEDOUT) the application may call **connect()** again for the same socket.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.

WSAEADDRINUSE	The socket's local address is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs at the time of **bind()**, but could be delayed until this function if the **bind()** was to a partially wild-card address (involving ADDR_ANY) and if a specific address needs to be "committed" at the time of this function.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	A non-blocking **connect()** call is in progress on the specified socket. **Important Note:** *In order to preserve backwards compatibility, this error is reported as WSAEINVAL to WinSock 1.1 applications that link to either WINSOCK.DLL or WSOCK32.DLL.*
WSAEADDRNOTAVAIL	The remote address is not a valid address (e.g., ADDR_ANY).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAEFAULT	The *name* or the *namelen* argument is not a valid part of the user address space, the *namelen* argument is too small, or the *name* argument contains incorrect address format for the associated address family.
WSAEINVAL	The parameter *s* is a listening socket, or the destination address specified is not consistent with that of the constrained group the socket belongs to.
WSAEISCONN	The socket is already connected (connection-oriented sockets only).
WSAENETUNREACH	The network can't be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAENOTSOCK	The descriptor is not a socket.

WSAETIMEDOUT	Attempt to connect timed out without establishing a connection.
WSAEWOULDBLOCK	The socket is marked as non-blocking and the connection cannot be completed immediately.
WSAEACCES	Attempt to connect datagram socket to broadcast address failed because **setsockopt()** SO_BROADCAST is not enabled.

See Also **accept(), bind(), getsockname(), socket(), select(), WSAAsyncSelect(), WSAConnect().**

4.5. getpeername()

Description Get the address of the peer to which a socket is connected.

```
#include <winsock2.h>

int WSAAPI
getpeername (
        IN          SOCKET                s,
        OUT         struct sockaddr FAR*  name,
        IN OUT      int FAR*              namelen
);
```

s	A descriptor identifying a connected socket.
name	The structure which is to receive the name of the peer.
namelen	A pointer to the size of the *name* structure.

Remarks **getpeername()** retrieves the name of the peer connected to the socket *s* and stores it in the struct sockaddr identified by *name*. It may be used only on a connected socket. For datagram sockets, only the name of a peer specified in a previous **connect()** call will be returned - any name specified by a previous **sendto()** call will not be returned by **getpeername()**.

On call, the *namelen* argument contains the size of the *name* buffer in bytes. On return, the *namelen* argument contains the actual size of the name returned in bytes.

Return Value If no error occurs, **getpeername()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *name* or the *namelen* argument is not a valid part of the user address space, or the *namelen* argument is too small.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.

See Also **bind()**, **socket()**, **getsockname()**.

4.6. getsockname()

Description Get the local name for a socket.

```
#include <winsock2.h>

int WSAAPI
getsockname (
        IN          SOCKET                  s,
        OUT         struct sockaddr FAR*    name,
        IN OUT      int FAR*                namelen
);
```

s	A descriptor identifying a bound socket.
name	Receives the address (name) of the socket.
namelen	The size of the *name* buffer.

Remarks **getsockname()** retrieves the current name for the specified socket descriptor in *name*. It is used on a bound and/or connected socket specified by the *s* parameter. The local association is returned. This call is especially useful when a **connect()** call has been made without doing a **bind()** first; this call provides the only means by which you can determine the local association which has been set by the system.

On call, the *namelen* argument contains the size of the *name* buffer in bytes. On return, the *namelen* argument contains the actual size of the name returned in bytes.

If a socket was bound to an unspecified address (e.g., ADDR_ANY), indicating that any of the host's addresses within the specified address family should be used for the socket, **getsockname()** will not necessarily return information about the host address, unless the socket has been connected with **connect()** or **accept().** A WinSock application must not assume that the address will be specified unless the socket is connected. This is because for a multi-homed host the address that will be used for the socket is unknown unless the socket is connected. If the socket is using a connectionless protocol, the address may not be available until I/O occurs on the socket.

Return Value If no error occurs, **getsockname()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *name* or the *namelen* argument is not a valid part of the user address space, or the *namelen* argument is too small.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.

WSAENOTSOCK	The descriptor is not a socket.
WSAEINVAL	The socket has not been bound to an address with **bind()**, or ADDR_ANY is specified in **bind()** but connection has not yet occurs**.**

See Also **bind()**, **socket(), getpeername()**.

4.7. getsockopt()

Description Retrieve a socket option.

```
#include <winsock2.h>

int WSAAPI
getsockopt (
        IN          SOCKET      s,
        IN          int         level,
        IN          int         optname,
        OUT         char FAR*   optval,
        IN OUT      int FAR*    optlen
);
```

s A descriptor identifying a socket.

level The level at which the option is defined; the supported *levels* include SOL_SOCKET and IPPROTO_TCP. (See annex for more protocol-specific *levels*.)

optname The socket option for which the value is to be retrieved.

optval A pointer to the buffer in which the value for the requested option is to be returned.

optlen A pointer to the size of the *optval* buffer.

Remarks **getsockopt()** retrieves the current value for a socket option associated with a socket of any type, in any state, and stores the result in *optval*. Options may exist at multiple protocol levels, but they are always present at the uppermost "socket" level. Options affect socket operations, such as the routing of packets, out-of-band data transfer, etc.

The value associated with the selected option is returned in the buffer *optval*. The integer pointed to by *optlen* should originally contain the size of this buffer; on return, it will be set to the size of the value returned. For SO_LINGER, this will be the size of a struct linger; for most other options it will be the size of an integer.

The application is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specified.

If the option was never set with **setsockopt(),** then **getsockopt()** returns the default value for the option.

The following options are supported for **getsockopt()**. The Type identifies the type of data addressed by *optval* (NOTE: there are other protocol-specific options available, as described in the ***Windows Sockets 2 Protocol Specific Annex***).

***Level* = SOL_SOCKET**			
Value	Type	Meaning	Default

SO_ACCEPTCONN	BOOL	Socket is **listen()**ing.	FALSE unless a **WSPListen()** has been performed
SO_BROADCAST	BOOL	Socket is configured for the transmission of broadcast messages.	FALSE
SO_DEBUG	BOOL	Debugging is enabled.	FALSE
SO_DONTLINGER	BOOL	If true, the SO_LINGER option is disabled.	TRUE
SO_DONTROUTE	BOOL	Routing is disabled.	FALSE
SO_ERROR	int	Retrieve error status and clear.	0
SO_GROUP_ID	GROUP	Reserved for future use with socket groups: The identifier of the group to which this socket belongs.	NULL
SO_GROUP_PRIORITY	int	Reserved for future use with socket groups: The relative priority for sockets that are part of a socket group.	0
SO_KEEPALIVE	BOOL	Keepalives are being sent.	FALSE
SO_LINGER	struct linger	Returns the current linger options.	l_onoff is 0
SO_MAX_MSG_SIZE	unsigned int	Maximum outbound (send) size of a message for message-oriented socket types (e.g., SOCK_DGRAM). There is no provision for finding out the maximum inbound message size. Has no meaning for stream-oriented sockets.	Implementation dependent
SO_OOBINLINE	BOOL	Out-of-band data is being received in the normal data stream. (See section 3.3.2. Winsock 1.1 Blocking routines & EINPROGRESS for a discussion of this topic.).	FALSE
SO_PROTOCOL_INFO	WSAPROTOCOL_INFO	Description of protocol info for protocol that is bound to this socket.	protocol dependent
SO_RCVBUF	int	Total per-socket buffer space reserved for receives. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.	Implementation dependent
SO_REUSEADDR	BOOL	The socket may be bound to an address which is already in use.	FALSE
SO_SNDBUF	int	Total per-socket buffer space reserved for sends. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.	Implementation dependent

SO_TYPE	int	The type of the socket (e.g. SOCK_STREAM).	As created via **socket()**
PVD_CONFIG	Service Provider Dependent	An "opaque" data structure object from the service provider associated with socket *s*. This object stores the current configuration information of the service provider. The exact format of this data structure is service provider specific.	Implementation dependent

***level* = IPPROTO_TCP**			
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.	Implementation dependent

BSD options not supported for **getsockopt()** are:

Value	Type	Meaning
SO_RCVLOWAT	int	Receive low water mark
SO_RCVTIMEO	int	Receive timeout
SO_SNDLOWAT	int	Send low water mark
SO_SNDTIMEO	int	Send timeout
TCP_MAXSEG	int	Get TCP maximum segment size

Calling **getsockopt()** with an unsupported option will result in an error code of WSAENOPROTOOPT being returned from **WSAGetLastError()**.

SO_DEBUG

WinSock service providers are encouraged (but not required) to supply output debug information if the SO_DEBUG option is set by an application. The mechanism for generating the debug information and the form it takes are beyond the scope of this specification.

SO_ERROR

The SO_ERROR option returns and resets the per-socket based error code, which is different from the per-thread based error code that is handled using the **WSAGetLastError()** and **WSASetLastError()** function calls. A successful call using the socket does not reset the socket based error code returned by the SO_ERROR option.

SO_GROUP_ID

Reserved for future use with socket groups: This is a get-only socket option which indicates the identifier of the group this socket belongs to. Note that socket group IDs are unique across all processes for a given service provider. If this socket is not a group socket, the value is NULL.

SO_GROUP_PRIORITY

Reserved for future use with socket groups: Group priority indicates the priority of the specified socket relative to other sockets within the socket group. Values are non-negative integers, with zero corresponding to the highest priority. Priority values represent a hint to the underlying service provider about how potentially scarce resources should be allocated. For example, whenever two or more sockets are both ready to transmit data, the highest priority socket (lowest value for SO_GROUP_PRIORITY) should be serviced first, with the remainder serviced in turn according to their relative priorities.

The WSAENOPROTOOPT error code is indicated for non group sockets or for service providers which do not support group sockets.

SO_KEEPALIVE

An application may request that a TCP/IP service provider enable the use of "keep-alive" packets on TCP-connections by turning on the SO_KEEPALIVE socket option. A WinSock provider need not support the use of keep-alive: if it does, the precise semantics are implementation-specific but should conform to section 4.2.3.6 of RFC 1122: *Requirements for Internet Hosts -- Communication Layers*. If a connection is dropped as the result of "keep-alives" the error code WSAENETRESET is returned to any calls in progress on the socket, and any subsequent calls will fail with WSAENOTCONN.

SO_LINGER

SO_LINGER controls the action taken when unsent data is queued on a socket and a **closesocket()** is performed. See **closesocket()** for a description of the way in which the SO_LINGER settings affect the semantics of **closesocket()**. The application gets the current behavior by retrieving a *struct linger* (pointed to by the *optval* argument) with the following elements:

```
struct linger {
      u_short     l_onoff;
      u_short     l_linger;
}
```

SO_MAX_MSG_SIZE

This is a get-only socket option which indicates the maximum outbound (send) size of a message for message-oriented socket types (e.g., SOCK_DGRAM) as implemented by a particular service provider. It has no meaning for byte stream oriented sockets. There is no provision to find out the maximum inbound message size.

SO_PROTOCOL_INFO

This is a get-only option which supplies the WSAPROTOCOL_INFO structure associated with this socket. See **WSAEnumProtocols()** for more information about this structure.

SO_RCVBUF
SO_SNDBUF

When a Windows Sockets implementation supports the SO_RCVBUF and SO_SNDBUF options, an application may request different buffer sizes (larger or smaller). The call to **setsockopt()** may succeed although the implementation did not provide the whole amount requested. An application must call this function with the same option to check the buffer size actually provided.

SO_REUSEADDR

By default, a socket may not be bound (see **bind()**) to a local address which is already in use. On occasions, however, it may be desirable to "re-use" an address in this way. Since every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different. To inform the WinSock provider that a **bind()** on a socket should not be disallowed because the desired address is already in use by another socket, the application should set the SO_REUSEADDR socket option for the socket before issuing the **bind()**. Note that the option is interpreted only at the time of the **bind()**: it is therefore unnecessary (but harmless) to set the option on a socket which is not to be bound to an existing address, and setting or resetting the option after the **bind()** has no effect on this or any other socket.

PVD_CONFIG

This option retrieves an "opaque" data structure object from the service provider associated with socket *s*. This object stores the current configuration information of the service provider. The exact format of this data structure is service provider specific.

TCP_NODELAY

The Nagle algorithm is disabled if the TCP_NODELAY option is enabled (and vice versa.). The Nagle algorithm (described in RFC 896) is very effective in reducing the number of small packets sent by a host by essentially buffering send data if there is unacknowledged data already "in flight" or until a full-size packet can be sent. It is highly recommended that Windows Sockets implementations enable the Nagle Algorithm by default, and for the vast majority of application protocols the Nagle Algorithm can deliver significant performance enhancements. However, for some applications this algorithm can impede performance, and **setsockopt()** with the same option may be used to turn it off. These are applications where many small messages are sent, which need to be received by the peer with the time delays between the messages maintained.

Return Value

If no error occurs, **getsockopt()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	One of the *optval* or the *optlen* arguments is not a valid part of the user address space, or the *optlen* argument is too small.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function).
WSAEINVAL	*level* is unknown or invalid
WSAENOPROTOOPT	The option is unknown or unsupported by the indicated protocol family.

WSAENOTSOCK	The descriptor is not a socket.

See Also **setsockopt()**, **socket(), WSAAsyncSelect(), WSAConnect(), WSAGetLastError(), WSASetLastError()**.

4.8. htonl()

Description Convert a **u_long** from host to TCP/IP network byte order (which is big-endian).

```
#include <winsock2.h>

u_long WSAAPI
htonl (
        IN      u_long  hostlong
);
```

hostlong A 32-bit number in host byte order.

Remarks This routine takes a 32-bit number in host byte order and returns a 32-bit number in the network byte order used in TCP/IP networks.

Return Value **htonl()** returns the value in TCP/IP's network byte order.

See Also **htons()**, **ntohl()**, **ntohs(), WSAHtons(), WSAHtonl(), WSANtohl(), WSANtohs()**.

4.9. htons()

Description Convert a **u_short** from host to TCP/IP network byte order (which is big-endian).

.

```
#include <winsock2.h>

u_short WSAAPI
htons (
        IN      u_short         hostshort
 );
```

hostshort A 16-bit number in host byte order.

Remarks This routine takes a 16-bit number in host byte order and returns a 16-bit number in network byte order used in TCP/IP networks.

Return Value **htons()** returns the value in TCP/IP network byte order.

See Also **htonl()**, **ntohl()**, **ntohs(), WSAHtons(), WSAHtonl(), WSANtohl(), WSANtohs()**.

4.10. ioctlsocket()

Description Control the mode of a socket.

```
#include <winsock2.h>

int WSAAPI
ioctlsocket (
        IN          SOCKET      s,
        IN          long        cmd,
        IN OUT      u_long FAR* argp
);
```

s A descriptor identifying a socket.

cmd The command to perform on the socket *s*.

argp A pointer to a parameter for *cmd*.

Remarks This routine may be used on any socket in any state. It is used to get or retrieve operating parameters associated with the socket, independent of the protocol and communications subsystem. The following commands are supported:

Command	Semantics
FIONBIO	Enable or disable non-blocking mode on socket *s*. *argp* points at an **unsigned long**, which is non-zero if non-blocking mode is to be enabled and zero if it is to be disabled. When a socket is created, it operates in blocking mode (i.e. non-blocking mode is disabled). This is consistent with BSD sockets. The **WSAAsyncSelect()** or **WSAEventSelect()** routine automatically sets a socket to nonblocking mode. If **WSAAsyncSelect()** or **WSAEventSelect()** has been issued on a socket, then any attempt to use **ioctlsocket()** to set the socket back to blocking mode will fail with WSAEINVAL. To set the socket back to blocking mode, an application must first disable **WSAAsyncSelect()** by calling **WSAAsyncSelect()** with the *lEvent* parameter equal to 0, or disable **WSAEventSelect()** by calling **WSAEventSelect()** with the *lNetworkEvents* parameter equal to 0.
FIONREAD	Determine the amount of data which can be read atomically from socket *s*. *argp* points to an **unsigned long** in which **ioctlsocket()** stores the result. If *s* is stream-oriented (e.g., type SOCK_STREAM), FIONREAD returns an amount of data which may be read in a single **recv()**; this may or may not be the same as the total amount of data queued on the socket. If *s* is message-oriented (e.g., type SOCK_DGRAM), FIONREAD returns the size of the first datagram (message) queued on the socket.
SIOCATMARK	Determine whether or not all out-of-band data has been read (See section *3.5.2 OOB Data in TCP* for a discussion of this topic.). This applies only to a socket of stream style (e.g., type SOCK_STREAM)

which has been configured for in-line reception of any out-of-band data (SO_OOBINLINE). If no out-of-band data is waiting to be read, the operation returns TRUE. Otherwise it returns FALSE, and the next **recv()** or **recvfrom()** performed on the socket will retrieve some or all of the data preceding the "mark"; the application should use the SIOCATMARK operation to determine whether any remains. If there is any normal data preceding the "urgent" (out of band) data, it will be received in order. (Note that a **recv()** or **recvfrom()** will never mix out-of-band and normal data in the same call.) *argp* points to an unsigned long in which **ioctlsocket()** stores the boolean result.

Compatibility

This function is a subset of **ioctl()** as used in Berkeley sockets. In particular, there is no command which is equivalent to FIOASYNC, while SIOCATMARK is the only socket-level command which is supported.

Return Value

Upon successful completion, the **ioctlsocket()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	*cmd* is not a valid command, or *argp* is not an acceptable parameter for *cmd*, or the command is not applicable to the type of socket supplied.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor *s* is not a socket.
WSAEFAULT	The *argp* argument is not a valid part of the user address space.

See Also

socket(), **setsockopt()**, **getsockopt()**, **WSAAsyncSelect(), WSAEventSelect(), WSAIoctl().**

4.11. listen()

Description Establish a socket to listen for incoming connection.

```
#include <winsock2.h>

int WSAAPI
listen (
        IN      SOCKET      s,
        IN      int         backlog
);
```

s A descriptor identifying a bound, unconnected socket.

backlog The maximum length to which the queue of pending connections may grow. If this value is SOMAXCONN, then the underlying service provider responsible for socket *s* will set the backlog to a maximum "reasonable" value. There is no standard provision to find out the actual backlog value used.

Remarks To accept connections, a socket is first created with **socket()**, bound to a local address with **bind()**, a backlog for incoming connections is specified with **listen()**, and then the connections are accepted with **accept()**. **listen()** applies only to sockets that are connection-oriented , e.g., those of type SOCK_STREAM. The socket *s* is put into "passive" mode where incoming connection requests are acknowledged and queued pending acceptance by the process.

This function is typically used by servers that could have more than one connection request at a time: if a connection request arrives with the queue full, the client will receive an error with an indication of WSAECONNREFUSED.

listen() attempts to continue to function rationally when there are no available descriptors. It will accept connections until the queue is emptied. If descriptors become available, a later call to **listen()** or **accept()** will re-fill the queue to the current or most recent "backlog", if possible, and resume listening for incoming connections.

An application may call **listen()** more than once on the same socket. This has the effect of updating the current backlog for the listening socket. Should there be more pending connections than the new *backlog* value, the excess pending connections will be reset and dropped.

Compatibility *backlog* is limited (silently) to a reasonable value as determined by the underlying service provider. Illegal values are replaced by the nearest legal value. There is no standard provision to find out the actual backlog value used.

Return Value If no error occurs, **listen()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.

WSAEADDRINUSE	The socket's local address is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs at the time of **bind()**, but could be delayed until this function if the **bind()** was to a partially wild-card address (involving ADDR_ANY) and if a specific address needs to be "committed" at the time of this function.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The socket has not been bound with **bind()**.
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The referenced socket is not of a type that supports the **listen()** operation.

See Also **accept()**, **connect()**, **socket()**.

4.12. ntohl()

Description Convert a **u_long** from TCP/IP network order to host byte order (which is big-endian).

```
#include <winsock2.h>

u_long WSAAPI
ntohl (
        IN      u_long          netlong
 );
```

netlong A 32-bit number in TCP/IP network byte order.

Remarks This routine takes a 32-bit number in TCP/IP network byte order and returns a 32-bit number in host byte order.

Return Value **ntohl()** returns the value in host byte order.

See Also **htonl()**, **htons()**, **ntohs(), WSAHtons(), WSAHtonl(), WSANtohl(), WSANtohs()**.

4.13. ntohs()

Description Convert a **u_short** from TCP/IP network byte order to host byte order (which is big-endian).

```
#include <winsock2.h>

u_short WSAAPI
ntohs (
        IN      u_short         netshort
 );
```

netshort A 16-bit number in TCP/IP network byte order.

Remarks This routine takes a 16-bit number in TCP/IP network byte order and returns a 16-bit number in host byte order.

Return Value **ntohs()** returns the value in host byte order.

See Also **htonl()**, **htons()**, **ntohl(), WSAHtons(), WSAHtonl(), WSANtohl(), WSANtohs()**.

4.14. recv()

Description Receive data from a connected socket.

#include <winsock2.h>

```
int WSAAPI
recv (
        IN      SOCKET          s,
        OUT     char FAR*       buf,
        IN      int             len,
        IN      int             flags
);
```

s	A descriptor identifying a connected socket.
buf	A buffer for the incoming data.
len	The length of *buf*.
flags	Specifies the way in which the call is made.

Remarks This function is used on connected sockets or bound connectionless sockets specified by the *s* parameter and is used to read incoming data. The socket's local address must be known. For server applications, this is usually done explicitly through **bind()** or implicitly through **accept()** or **WSAAccept()**. Explicit binding is discouraged for client applications. For client applications the socket can become bound implicitly to a local address through **connect(), WSAConnect(), sendto(), WSASendTo(),** or **WSAJoinLeaf()**.

For connected, connectionless sockets, this function restricts the addresses from which received messages are accepted. The function only returns messages from the remote address specified in the connection. Messages from other addresses are (silently) discarded.

For byte stream style socket (e.g., type SOCK_STREAM), as much information as is currently available up to the size of the buffer supplied is returned. If the socket has been configured for in-line reception of out-of-band data (socket option SO_OOBINLINE) and out-of-band data is unread, only out-of-band data will be returned. The application may use the **ioctlsocket()** or **WSAIoctl()** with the SIOCATMARK command to determine whether any more out-of-band data remains to be read.

For message-oriented sockets (e.g., type SOCK_DGRAM), data is extracted from the first enqueued datagram (message) from the destination address specified in the **connect()** call. If the datagram or message is larger than the buffer supplied, the buffer is filled with the first part of the datagram, and **recv()** generates the error WSAEMSGSIZE. For unreliable protocols (e.g. UDP) the excess data is lost, for reliable protocols the data is retained by the service provider until it is successfully read by calling **recv()** with a large enough buffer. For TCP/IP, an application cannot receive from any multicast address until after becoming a group member (see ***Windows Sockets 2 Protocol-Specific Annex*** for more information).

If no incoming data is available at the socket, the **recv()** call blocks and waits for data to arrive according to the blocking rules defined for **WSARecv()** with the MSG_PARTIAL flag not set unless the socket is non-blocking. In this case a value of SOCKET_ERROR is returned with the error code set to WSAEWOULDBLOCK. The **select()**, **WSAAsyncSelect()**, or **WSAEventSelect()** calls may be used to determine when more data arrives.

If the socket is connection-oriented and the remote side has shut down the connection gracefully, and all data has been received already, a **recv()** will complete immediately with 0 bytes received. If the connection has been reset, a **recv()** will fail with the error WSAECONNRESET.

Flags may be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
MSG_PEEK	Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue, and the function returns the number of bytes currently pending to receive.
MSG_OOB	Process out-of-band data (See section *3.5. Out-Of-Band data* for a discussion of this topic.)

Return Value

If no error occurs, **recv()** returns the number of bytes received. If the connection has been gracefully closed, and all data received, the return value is 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *buf* argument is not totally contained in a valid part of the user address space.
WSAENOTCONN	The socket is not connected.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENETRESET	The connection has been broken due to “keep-alive” activity detecting a failure while the operation was in progress.
WSAENOTSOCK	The descriptor is not a socket.

WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **recv()** on a socket after **shutdown()** has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as non-blocking and the receive operation would block.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.
WSAEINVAL	The socket has not been bound (e.g., with **bind()**), or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled or (for byte stream sockets only) *len* was 0 or negative**.**
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure. The application should close the socket as it is no longer useable.
WSAETIMEDOUT	The connection has been dropped because of a network failure or because the peer system failed to respond.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a “hard” or “abortive” close. The application should close the socket as it is no longer useable. On a UDP datagram socket this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.

See Also **recvfrom()**, **send()**, **select()**, **socket()**, **WSAAsyncSelect().**

4.15. recvfrom()

Description Receive a datagram and store the source address.

```
#include <winsock2.h>

int WSAAPI
recvfrom (
        IN          SOCKET                s,
        OUT         char FAR*             buf,
        IN          int                   len,
        IN          int                   flags,
        OUT         struct sockaddr FAR*  from,
        IN OUT      int FAR*              fromlen
);
```

s	A descriptor identifying a bound socket.
buf	A buffer for the incoming data.
len	The length of *buf*.
flags	Specifies the way in which the call is made.
from	An optional pointer to a buffer which will hold the source address upon return.
fromlen	An optional pointer to the size of the *from* buffer.

Remarks This function is used to read incoming data on a socket and capture the address from which the data was sent. The socket must not be connected. The socket's local address must be known. For server applications, this is usually done explicitly through **bind()**. Explicit binding is discouraged for client applications. For client applications using this function, the socket can become bound implicitly to a local address through **sendto(), WSASendTo(),** or **WSAJoinLeaf()**.

For stream-oriented sockets such as those of type SOCK_STREAM, as much information as is currently available up to the size of the buffer supplied is returned. If the socket has been configured for in-line reception of out-of-band data (socket option SO_OOBINLINE) and out-of-band data is unread, only out-of-band data will be returned. The application may use the **ioctlsocket()** or **WSAIoctl()** SIOCATMARK command to determine whether any more out-of-band data remains to be read. The *from* and *fromlen* parameters are ignored for connection-oriented sockets.

For message-oriented sockets, data is extracted from the first enqueued message, up to the size of the buffer supplied. If the datagram or message is larger than the buffer supplied, the buffer is filled with the first part of the datagram, and **recvfrom()** generates the error WSAEMSGSIZE. For unreliable protocols (e.g. UDP) the excess data is lost.

If *from* is non-zero, and the socket is not connection-oriented (e.g., type SOCK_DGRAM), the network address of the peer which sent the data is copied to the corresponding struct sockaddr. The value pointed to by *fromlen* is initialized to the size

of this structure, and is modified on return to indicate the actual size of the address stored there.

If no incoming data is available at the socket, the **recvfrom()** call blocks and waits for data to arrive according to the blocking rules defined for **WSARecv()** with the MSG_PARTIAL flag not set unless the socket is non-blocking. In this case a value of SOCKET_ERROR is returned with the error code set to WSAEWOULDBLOCK. The **select()**, **WSAAsyncSelect()**, or **WSAEventSelect()** may be used to determine when more data arrives.

If the socket is connection-oriented and the remote side has shut down the connection gracefully, a **recvfrom()** will complete immediately with 0 bytes received. If the connection has been reset **recvfrom()** will fail with the error WSAECONNRESET.

Flags may be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
MSG_PEEK	Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue, and the function returns the number of bytes currently pending to receive
MSG_OOB	Process out-of-band data (See section *3.5. Out-Of-Band data* for a discussion of this topic.)

Return Value

If no error occurs, **recvfrom()** returns the number of bytes received. If the connection has been gracefully closed, and all data received, the return value is 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *buf* or *from* parameters are not part of the user address space, or the *fromlen* argument is too small to accommodate the peer address.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.

WSAEINVAL	The socket has not been bound (e.g., with **bind()**), or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled, or (for byte stream style sockets only) *len* was 0 or negative.
WSAEISCONN	The socket is connected. This function is not permitted with a connected socket, whether the socket is connection-oriented or connectionless.
WSAENETRESET	The connection has been broken due to "keep-alive" activity detecting a failure while the operation was in progress.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **recvfrom()** on a socket after **shutdown()** has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as non-blocking and the **recvfrom()** operation would block.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.
WSAETIMEDOUT	The connection has been dropped, because of a network failure or because the system on the other end went down without notice.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a "hard" or "abortive" close. The application should close the socket as it is no longer useable. On a UDP datagram socket this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.

See Also **recv()**, **send()**, **socket()**, **WSAAsyncSelect(), WSAEventSelect().**

4.16. select()

Description Determine the status of one or more sockets, waiting if necessary.

```
#include <winsock2.h>

int WSAAPI
select (
        IN          int                 nfds,
        IN OUT      fd_set FAR *        readfds,
        IN OUT      fd_set FAR *        writefds,
        IN OUT      fd_set FAR *        exceptfds,
        IN          const struct timeval FAR *      timeout
);
```

nfds	This argument is ignored and included only for the sake of compatibility.
readfds	An optional pointer to a set of sockets to be checked for readability.
writefds	An optional pointer to a set of sockets to be checked for writability
exceptfds	An optional pointer to a set of sockets to be checked for errors.
timeout	The maximum time for **select()** to wait, or NULL for blocking operation.

Remarks This function is used to determine the status of one or more sockets. For each socket, the caller may request information on read, write or error status. The set of sockets for which a given status is requested is indicated by an fd_set structure. The sockets contained within the fd_set structures must be associated with a single service provider. For the purpose of this restriction, sockets are considered to be from the same service provider if the WSAPROTOCOL_INFO structures describing their protocols have the same *providerId* value.

Upon return, the structures are updated to reflect the subset of these sockets which meet the specified condition, and **select()** returns the number of sockets meeting the conditions. A set of macros is provided for manipulating an fd_set. These macros are compatible with those used in the Berkeley software, but the underlying representation is completely different.

The parameter *readfds* identifies those sockets which are to be checked for readability. If the socket is currently **listen()**ing, it will be marked as readable if an incoming connection request has been received, so that an **accept()** is guaranteed to complete without blocking. For other sockets, readability means that queued data is available for reading so that a **recv()** or **recvfrom()**, **WSARecv()** or **WSARecvFrom()**,is guaranteed not to block.

For connection-oriented sockets, readability may also indicate that a close request has been received from the peer. If the virtual circuit was closed gracefully, and all data received, then a **recv()** will return immediately with 0 bytes read. If the virtual circuit was reset, then a **recv()** will complete immediately with an error code, such as

WSAECONNRESET. The presence of out-of-band data will be checked if the socket option SO_OOBINLINE has been enabled (see **setsockopt()**).

The parameter *writefds* identifies those sockets which are to be checked for writability. If a socket is **connect()**ing (non-blocking), writability means that the connection establishment successfully completed. If the socket is not in the process of **connect()**ing, writability means that a **send()** or **sendto(), or WSASend() or WSASendto(),** are guaranteed to succeed. However, they may block on a blocking socket if the *len* exceeds the amount of outgoing system buffer space available. [It is not specified how long these guarantees can be assumed to be valid, particularly in a multithreaded environment.]

The parameter *exceptfds* identifies those sockets which are to be checked for the presence of out-of-band data (see section *3.5. Out-Of-Band data* for a discussion of this topic) or any exceptional error conditions. Note that out-of-band data will only be reported in this way if the option SO_OOBINLINE is FALSE. If a socket is **connect()**ing (non-blocking), failure of the connect attempt is indicated in *exceptfds* (application must then call **getsockopt()** SO_ERROR to determine the error value to describe why the failure occurred. This specification does not define which other errors will be included.

Any two of *readfds*, *writefds*, or *exceptfds* may be given as NULL if no descriptors are to be checked for the condition of interest. At least one must be non-NULL, and any non-NULL descriptor set must contain at least one socket descriptor..

Summary: A socket will be identified in a particular set when **select()** returns if:

readfds:
* If **listen()**ing, a connection is pending, **accept()** will succeed
* Data is available for reading (includes OOB data if SO_OOBINLINE is enabled)
* Connection has been closed/reset/aborted

writefds:
* If **connect()**ing (non-blocking), connection has succeeded.
* Data may be sent

exceptfds:
* If **connect()**ing (non-blocking), connection attempt failed.
* OOB data is available for reading (only if SO_OOBINLINE is disabled)

Four macros are defined in the header file **winsock2.h** for manipulating and checking the descriptor sets. The variable FD_SETSIZE determines the maximum number of descriptors in a set. (The default value of FD_SETSIZE is 64, which may be modified by #defining FD_SETSIZE to another value before #including **winsock2.h**.) Internally, socket handles in a fd_set are not represented as bit flags as in Berkeley Unix. Their data representation is opaque. Use of these macros will maintain software portability between different socket environments. The macros to manipulate and check fd_set contents are:

FD_CLR(*s***, ****set***)** Removes the descriptor *s* from *set*.

FD_ISSET(*s***, ****set***)** Nonzero if *s* is a member of the *set*, zero otherwise.

FD_SET(*s***, ****set***)** Adds descriptor *s* to *set*.

FD_ZERO(*_set_) Initializes the *set* to the NULL set.

The parameter *timeout* controls how long the **select()** may take to complete. If *timeout* is a null pointer, **select()** will block indefinitely until at least one descriptor meets the specified criteria. Otherwise, *timeout* points to a struct timeval which specifies the maximum time that **select()** should wait before returning. When **select()** returns, the contents of the struct timeval are not altered. If the timeval is initialized to {0, 0}, **select()** will return immediately; this is used to "poll" the state of the selected sockets. If this is the case, then the **select()** call is considered nonblocking and the standard assumptions for nonblocking calls apply. For example, the blocking hook will not be called, and WinSock will not yield.

Return Value

select() returns the total number of descriptors which are ready and contained in the fd_set structures, 0 if the time limit expired, or SOCKET_ERROR if an error occurred. If the return value is SOCKET_ERROR, **WSAGetLastError()** may be used to retrieve a specific error code.

Comments

select() has no effect on the persistence of socket events registered with **WSAAsyncSelect()** or **WSAEventSelect()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAEFAULT	The WinSock implementation was unable to allocate needed resources for its internal operations, or the *readfds*, *writefds*, *exceptfds*, or *timeval* parameters are not part of the user address space.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	The *timeout* value is not valid, or all three descriptor parameters were NULL.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	One of the descriptor sets contains an entry which is not a socket.

See Also

accept(), **connect()**, **recv()**, **recvfrom()**, **send(), WSAAsyncSelect(), WSAEventSelect()**

4.17. send()

Description Send data on a connected socket.

```
#include <winsock2.h>

int WSAAPI
send (
        IN      SOCKET              s,
        IN      const char FAR *    buf,
        IN      int                 len,
        IN      int                 flags
);
```

s A descriptor identifying a connected socket.

buf A buffer containing the data to be transmitted.

len The length of the data in *buf*.

flags Specifies the way in which the call is made.

Remarks **send()** is used to write outgoing data on a connected socket. For message-oriented sockets, care must be taken not to exceed the maximum packet size of the underlying provider, which can be obtained by getting the value of socket option **SO_MAX_MSG_SIZE**. If the data is too long to pass atomically through the underlying protocol the error WSAEMSGSIZE is returned, and no data is transmitted.

Note that the successful completion of a **send()** does not indicate that the data was successfully delivered.

If no buffer space is available within the transport system to hold the data to be transmitted, **send()** will block unless the socket has been placed in a non-blocking I/O mode. On non-blocking stream-oriented sockets, the number of bytes written may be between 1 and the requested length, depending on buffer availability on both the local and foreign hosts. The **select()**, **WSAAsyncSelect()** or **WSAEventSelect()** call may be used to determine when it is possible to send more data.

Calling **send()** with a *len* of 0 is to be treated by implementations as successful - in this case **send()** may return 0 as a valid return value. For message-oriented sockets, a zero-length transport datagram is sent.

Flags may be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A WinSock service provider may choose to ignore this flag.

MSG_OOB	Send out-of-band data (stream style socket such as SOCK_STREAM only).

Return Value If no error occurs, **send()** returns the total number of bytes sent. (Note that this may be less than the number indicated by *len* for non-blocking sockets.) Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set (call setsockopt SO_BROADCAST to allow use of the broadcast address).
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *buf* argument is not totally contained in a valid part of the user address space.
WSAENETRESET	The connection has been broken due to “keep-alive” activity detecting a failure while the operation was in progress.
WSAENOBUFS	No buffer space is available.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **send()** on a socket after **shutdown()** has been invoked with how set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as non-blocking and the requested operation would block.

WSAEMSGSIZE	The socket is message-oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEHOSTUNREACH	The remote host can't be reached from this host at this time.
WSAEINVAL	The socket has not been bound with **bind()**, or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled**.**
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure. The application should close the socket as it is no longer useable.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a “hard” or “abortive” close. For UPD sockets, the remote host was unable to deliver a previously sent UDP datagram and responded with a "Port Unreachable" ICMP packet. The application should close the socket as it is no longer useable.
WSAETIMEDOUT	The connection has been dropped, because of a network failure or because the system on the other end went down without notice.

See Also **recv()**, **recvfrom()**, **select()**, **socket()**, **sendto()**, **WSAAsyncSelect(), WSAEventSelect()**.

4.18. sendto()

Description Send data to a specific destination.

```
#include <winsock2.h>

int WSAAPI
sendto (
        IN      SOCKET                      s,
        IN      const char FAR *            buf,
        IN      int                         len,
        IN      int                         flags,
        IN      const struct sockaddr FAR * to,
        IN      int                         tolen
 );
```

s	A descriptor identifying a (possibly connected) socket.
buf	A buffer containing the data to be transmitted.
len	The length of the data in *buf*.
flags	Specifies the way in which the call is made.
to	An optional pointer to the address of the target socket.
tolen	The size of the address in *to*.

Remarks **sendto()** is used to write outgoing data on a socket. For message-oriented sockets, care must be taken not to exceed the maximum packet size of the underlying subnets, which can be obtained by getting the value of socket option SO_MAX_MSG_SIZE. If the data is too long to pass atomically through the underlying protocol the error WSAEMSGSIZE is returned, and no data is transmitted.

The *to* parameter may be any valid address in the socket's address family, including a broadcast or any multicast address. To send to a broadcast address, an application must have **setsockopt()** SO_BROADCAST enabled, otherwise **sendto()** will fail with the error code WSAEACCES. For TCP/IP, an application can send to any multicast address (without becoming a group member).

If the socket is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound. An application may use **getsockname()** to determine the local socket name in this case.

Note that the successful completion of a **sendto()** does not indicate that the data was successfully delivered.

sendto() is normally used on a connectionless socket to send a datagram to a specific peer socket identified by the *to* parameter. Even if the connectionless socket has been previously **connect()**ed to a specific address, *to* overrides the destination address for that particular datagram only. On a connection-oriented socket, the *to* and *tolen* parameters are ignored; in this case the **sendto()** is equivalent to **send()**.

For sockets using IP (version 4):
To send a broadcast (on a SOCK_DGRAM only), the address in the *to* parameter should be constructed using the special IP address INADDR_BROADCAST (defined in **winsock2.h**) together with the intended port number. It is generally inadvisable for a broadcast datagram to exceed the size at which fragmentation may occur, which implies that the data portion of the datagram (excluding headers) should not exceed 512 bytes.

If no buffer space is available within the transport system to hold the data to be transmitted, **sendto()** will block unless the socket has been placed in a non-blocking I/O mode. On non-blocking stream-oriented sockets, the number of bytes written may be between 1 and the requested length, depending on buffer availability on both the local and foreign hosts. The **select()**, **WSAAsyncSelect()** or **WSAEventSelect()** call may be used to determine when it is possible to send more data.

Calling **sendto()** with a *len* of 0 is legal and in this case **sendto()** will return 0 as a valid return value. For message-oriented sockets, a zero-length transport datagram is sent.

Flags may be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A WinSock service provider may choose to ignore this flag.
MSG_OOB	Send out-of-band data (stream style socket such as SOCK_STREAM only).

Return Value

If no error occurs, **sendto()** returns the total number of bytes sent. (Note that this may be less than the number indicated by *len*.) Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set (call setsockopt SO_BROADCAST to allow use of the broadcast address).
.	
WSAEINVAL	An unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**

WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *buf* or *to* parameters are not part of the user address space, or the *tolen* argument is too small.
WSAENETRESET	The connection has been broken due to “keep-alive” activity detecting a failure while the operation was in progress.
WSAENOBUFS	No buffer space is available.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only)
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **sendto()** on a socket after **shutdown()** has been invoked with *how* set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as non-blocking and the requested operation would block.
WSAEMSGSIZE	The socket is message-oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEHOSTUNREACH	The remote host can't be reached from this host at this time.
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure. The application should close the socket as it is no longer useable.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a “hard” or “abortive” close. For UPD sockets, the remote host was unable to deliver a previously sent UDP datagram and responded with a "Port Unreachable" ICMP packet. The application should close the socket as it is no longer useable.
WSAEADDRNOTAVAIL	The remote address is not a valid address (e.g., ADDR_ANY).

WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAEDESTADDRREQ	A destination address is required.
WSAENETUNREACH	The network can't be reached from this host at this time.
WSAETIMEDOUT	The connection has been dropped, because of a network failure or because the system on the other end went down without notice.

See Also **recv()**, **recvfrom()**, **select()**, **socket()**, **send()**, **WSAAsyncSelect(), WSAEventSelect()**.

4.19. setsockopt()

Description Set a socket option.

```
#include <winsock2.h>

int WSAAPI
setsockopt (
        IN      SOCKET          s,
        IN      int             level,
        IN      int             optname,
        IN      const char FAR *optval,
        IN      int             optlen
);
```

s	A descriptor identifying a socket.
level	The level at which the option is defined; the supported *level*s include SOL_SOCKET and IPPROTO_TCP. (See annex for more protocol-specific *levels*.)
optname	The socket option for which the value is to be set.
optval	A pointer to the buffer in which the value for the requested option is supplied.
optlen	The size of the *optval* buffer.

Remarks **setsockopt()** sets the current value for a socket option associated with a socket of any type, in any state. Although options may exist at multiple protocol levels, they are always present at the uppermost "socket" level. Options affect socket operations, such as whether expedited data is received in the normal data stream, whether broadcast messages may be sent on the socket, etc.

There are two types of socket options: Boolean options that enable or disable a feature or behavior, and options which require an integer value or structure. To enable a Boolean option, *optval* points to a nonzero integer. To disable the option *optval* points to an integer equal to zero. *optlen* should be equal to sizeof(int) for Boolean options. For other options, *optval* points to the an integer or structure that contains the desired value for the option, and *optlen* is the length of the integer or structure.

The following options are supported for **setsockopt()**. For default values of these options, see the description of **getsockopt()**. The Type identifies the type of data addressed by *optval*.

***Level* = SOL_SOCKET**		
Value	Type	Meaning
SO_BROADCAST	BOOL	Allow transmission of broadcast messages on the socket.
SO_DEBUG	BOOL	Record debugging information.
SO_DONTLINGER	BOOL	Don't block close waiting for unsent data to be sent. Setting this option is equivalent to setting SO_LINGER with *l_onoff* set to zero.

SO_DONTROUTE	BOOL	Don't route: send directly to interface.
SO_GROUP_PRIORITY	int	Reserved for future use with socket groups: Specify the relative priority to be established for sockets that are part of a socket group.
SO_KEEPALIVE	BOOL	Send keepalives
SO_LINGER	struct linger	Linger on close if unsent data is present
SO_OOBINLINE	BOOL	Receive out-of-band data in the normal data stream (see section *3.5. Out-Of-Band data* for a discussion of this topic).
SO_RCVBUF	int	Specify the total per-socket buffer space reserved for receives. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.
SO_REUSEADDR	BOOL	Allow the socket to be bound to an address which is already in use. (See **bind()**.)
SO_SNDBUF	int	Specify the total per-socket buffer space reserved for sends. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.
PVD_CONFIG	Service Provider Dependent	This object stores the configuration information for the service provider associated with socket *s*. The exact format of this data structure is service provider specific.

Level* = IPPROTO_TCP		
TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.

*included for backwards compatibility with WinSock 1.1

BSD options not supported for **setsockopt()** are:

Value	Type	Meaning
SO_ACCEPTCONN	BOOL	Socket is listening
SO_RCVLOWAT	int	Receive low water mark
SO_RCVTIMEO	int	Receive timeout
SO_SNDLOWAT	int	Send low water mark
SO_SNDTIMEO	int	Send timeout
SO_TYPE	int	Type of the socket

SO_DEBUG

WinSock service providers are encouraged (but not required) to supply output debug information if the SO_DEBUG option is set by an application. The mechanism for generating the debug information and the form it takes are beyond the scope of this specification.

SO_GROUP_PRIORITY

Reserved for future use with socket groups: Group priority indicates the relative priority of the specified socket relative to other sockets within the socket group. Values are non-negative integers, with zero corresponding to the highest priority. Priority values

represent a hint to the underlying service provider about how potentially scarce resources should be allocated. For example, whenever two or more sockets are both ready to transmit data, the highest priority socket (lowest value for SO_GROUP_PRIORITY) should be serviced first, with the remainder serviced in turn according to their relative priorities.

The WSAENOPROTOOPT error is indicated for non group sockets or for service providers which do not support group sockets.

SO_KEEPALIVE

An application may request that a TCP/IP provider enable the use of "keep-alive" packets on TCP-connections by turning on the SO_KEEPALIVE socket option. A WinSock provider need not support the use of keep-alives: if it does, the precise semantics are implementation-specific but should conform to section 4.2.3.6 of RFC 1122: *Requirements for Internet Hosts -- Communication Layers*. If a connection is dropped as the result of "keep-alives" the error code WSAENETRESET is returned to any calls in progress on the socket, and any subsequent calls will fail with WSAENOTCONN.

SO_LINGER

SO_LINGER controls the action taken when unsent data is queued on a socket and a **closesocket()** is performed. See **closesocket()** for a description of the way in which the SO_LINGER settings affect the semantics of **closesocket()**. The application sets the desired behavior by creating a *struct linger* (pointed to by the *optval* argument) with the following elements:

```
struct linger {
      u_short     l_onoff;
      u_short     l_linger;
}
```

To enable SO_LINGER, the application should set *l_onoff* to a non-zero value, set *l_linger* to 0 or the desired timeout (in seconds), and call **setsockopt()**. To enable SO_DONTLINGER (i.e. disable SO_LINGER) *l_onoff* should be set to zero and **setsockopt()** should be called. Note that enabling SO_LINGER with a non-zero timeout on a non-blocking socket is not recommended (see section 4.3. for details).

Enabling SO_LINGER also disables SO_DONTLINGER, and vice versa. Note that if SO_DONTLINGER is DISABLED (i.e. SO_LINGER is ENABLED) then no timeout value is specified. In this case the timeout used is implementation dependent. If a previous timeout has been established for a socket (by enabling SO_LINGER), then this timeout value should be reinstated by the service provider.

SO_REUSEADDR

By default, a socket may not be bound (see **bind()**) to a local address which is already in use. On occasions, however, it may be desirable to "re-use" an address in this way. Since every connection is uniquely identified by the combination of local and remote addresses, there is no problem with having two sockets bound to the same local address as long as the remote addresses are different. To inform the WinSock provider that a **bind()** on a socket should not be disallowed because the desired address is already in use by another socket, the application should set the SO_REUSEADDR socket option for the socket before issuing the **bind()**. Note that the option is interpreted only at the time of the **bind()**: it is therefore unnecessary (but harmless) to set the option on a

socket which is not to be bound to an existing address, and setting or resetting the option after the **bind()** has no effect on this or any other socket.

SO_RCVBUF
SO_SNDBUF

When a Windows Sockets implementation supports the SO_RCVBUF and SO_SNDBUF options, an application may request different buffer sizes (larger or smaller). The call to **setsockopt()** may succeed although the implementation did not provide the whole amount requested. An application must call **getsockopt()** with the same option to check the buffer size actually provided.

PVD_CONFIG

This object stores the configuration information for the service provider associated with socket *s*. The exact format of this data structure is service provider specific.

TCP_NODELAY

The TCP_NODELAY option is specific to TCP/IP service providers. Enabling the TCP_NODELAY option disables the TCP Nagle Algorithm (and vice versa). The Nagle algorithm (described in RFC 896) is very effective in reducing the number of small packets sent by a host by essentially buffering send data if there is unacknowledged data already "in flight" or until a full-size packet can be sent. It is highly recommended that TCP/IP service providers enable the Nagle Algorithm by default, and for the vast majority of application protocols the Nagle Algorithm can deliver significant performance enhancements. However, for some applications this algorithm can impede performance, and TCP_NODELAY may be used to turn it off. These are applications where many small messages are sent, which need to be received by the peer with the time delays between the messages maintained. Application writers should not set TCP_NODELAY unless the impact of doing so is well-understood and desired, since setting TCP_NODELAY can have a significant negative impact of network and application performance.

Return Value

If no error occurs, **setsockopt()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	*optval* is not in a valid part of the process address space or *optlen* argument is too small.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function (see section.
WSAEINVAL	*level* is not valid, or the information in *optval* is not valid.
WSAENETRESET	The connection has been broken due to "keep-alive" activity detecting a failure while the operation was in progress.

WSAENOPROTOOPT	The option is unknown or unsupported for the specified provider or socket (see SO_GROUP_PRIORITY limitations).
WSAENOTCONN	Connection has been reset when SO_KEEPALIVE is set.
WSAENOTSOCK	The descriptor is not a socket.

See Also **bind()**, **getsockopt()**, **ioctlsocket()**, **socket()**, **WSAAsyncSelect(), WSAEventSelect()**.

4.20. shutdown()

Description Disable sends and/or receives on a socket.

```
#include <winsock2.h>

int WSAAPI
shutdown (
        IN              SOCKET      s,
        IN              int         how
);
```

s	A descriptor identifying a socket.
how	A flag that describes what types of operation will no longer be allowed.

Remarks **shutdown()** is used on all types of sockets to disable reception, transmission, or both.

If *how* is SD_RECEIVE, subsequent receives on the socket will be disallowed. This has no effect on the lower protocol layers. For TCP sockets, if there is still data queued on the socket waiting to be received, or data arrives subsequently, the connection is reset, since the data cannot be delivered to the user. For UDP sockets, incoming datagrams are accepted and queued. In no case will an ICMP error packet be generated.

If *how* is SD_SEND, subsequent sends are disallowed. For TCP sockets, a FIN will be sent after all data is sent and acknowledged by the receiver..

Setting *how* to SD_BOTH disables both sends and receives as described above.

Note that **shutdown()** does not close the socket, and resources attached to the socket will not be freed until **closesocket()** is invoked.

To assure that all data is sent and received on a connected socket before it is closed, an application should use **shutdown()** to close connection before calling **closesocket()**. For example, to initiate a graceful disconnect, an application could:

- call **WSAAsyncSelect()** to register for FD_CLOSE notification
- call **shutdown()** with *how*=SD_SEND
- when FD_CLOSE received, call **recv()** until 0 returned, or SOCKET_ERROR
- call **closesocket()**

Comments **shutdown()** does not block regardless of the SO_LINGER setting on the socket.

An application should not rely on being able to re-use a socket after it has been shut down. In particular, a WinSock provider is not required to support the use of **connect()** on such a socket.

Return Value If no error occurs, **shutdown()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	*how* is not valid, or is not consistent with the socket type, e.g., SD_SEND is used with a UNI_RECV socket type.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.

See Also **connect()**, **socket()**.

4.21. socket()

Description Create a socket which is bound to a specific service provider.

```
#include <winsock2.h>

SOCKET WSAAPI
socket (
        IN          int         af,
        IN          int         type,
        IN          int         protocol
);
```

af An address family specification.

type A type specification for the new socket.

protocol A particular protocol to be used with the socket which is specific to the indicated address family.

Remarks **socket()** causes a socket descriptor and any related resources to be allocated and bound to a specific transport service provider. WinSock will utilize the first available service provider that supports the requested combination of address family, socket type and protocol parameters. The socket created will have the overlapped attribute by default. Note that on Microsoft operating systems there is a Microsoft-specific socket option, SO_OPENTYPE, defined in “mswsock.h” that can affect this default. See Microsoft-specific documentation for a detailed description of SO_OPENTYPE. Sockets without the overlapped attribute can be created by using **WSASocket()**. All functions that allow overlapped operation (**WSASend(), WSARecv(),WSASendTo(), WSARecvFrom(), WSAIoctl()**) also support non-overlapped usage on an overlapped socket if the values for parameters related to overlapped operation are NULL.

When selecting a protocol and its supporting service provider this procedure will only choose a base protocol or a protocol chain, not a protocol layer by itself. Unchained protocol layers are not considered to have "partial matches" on *type* or *af* either. That is, they do not lead to an error code of WSAEAFNOSUPPORT or WSAEPROTONOSUPPORT if no suitable protocol is found.

Note: the manifest constant AF_UNSPEC continues to be defined in the header file but its use is **strongly discouraged**, as this may cause ambiguity in interpreting the value of the *protocol* parameter.

The following are the only two *type* specifications supported for WinSock 1.1:

Type	Explanation
SOCK_STREAM	Provides sequenced, reliable, two-way, connection-based byte streams with an out-of-band data transmission mechanism. Uses TCP for the Internet address family.
SOCK_DGRAM	Supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small)

maximum length. Uses UDP for the Internet address family.

In WinSock 2 many new socket types will be introduced. However, since an application can dynamically discover the attributes of each available transport protocol via the **WSAEnumProtocols()** function, the various socket types need not be called out in the API specification. Socket type definitions will appear in Winsock2.h which will be periodically updated as new socket types, address families and protocols are defined.

Connection-oriented sockets such as SOCK_STREAM provide full-duplex connections, and must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a **connect()** call. Once connected, data may be transferred using **send()** and **recv()** calls. When a session has been completed, a **closesocket()** must be performed.

The communications protocols used to implement a reliable, connection-oriented socket ensure that data is not lost or duplicated. If data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and subsequent calls will fail with the error code set to WSAETIMEDOUT.

Connectionless, message-oriented sockets allow sending and receiving of datagrams to and from arbitrary peers using **sendto()** and **recvfrom()**. If such a socket is **connect()**ed to a specific peer, datagrams may be sent to that peer using **send()** and may be received from (only) this peer using **recv()**.

Support for sockets with type RAW is not required but service providers are encouraged to support raw sockets whenever it makes sense to do so.

Return Value

If no error occurs, **socket()** returns a descriptor referencing the new socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem or the associated service provider has failed.
WSAEAFNOSUPPORT	The specified address family is not supported.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAEPROTONOSUPPORT	The specified protocol is not supported.

WSAEPROTOTYPE	The specified protocol is the wrong type for this socket.
WSAESOCKTNOSUPPORT	The specified socket type is not supported in this address family.

See Also **accept()**, **bind()**, **connect()**, **getsockname()**, **getsockopt()**, **setsockopt()**, **listen()**, r**ecv()**, **recvfrom()**, **select()**, **send()**, **sendto()**, **shutdown()**, **ioctlsocket(), WSASocket()**.

4.22. WSAAccept()

Description Conditionally accept a connection based on the return value of a condition function, optionally create and/or join a socket group, provide QOS flowspecs, and allow transfer of connection data.

```
#include <winsock2.h>

SOCKET WSAAPI
WSAAccept (
        IN          SOCKET                s,
        OUT         struct sockaddr FAR * addr,
        IN OUT      LPINT                 addrlen,
        IN          LPCONDITIONPROC       lpfnCondition,
        IN          DWORD                 dwCallbackData
);
```

s A descriptor identifying a socket which is listening for connections after a **listen()**.

addr An optional pointer to a buffer which receives the address of the connecting entity, as known to the communications layer. The exact format of the *addr* argument is determined by the address family established when the socket was created.

addrlen An optional pointer to an integer which contains the length of the address *addr*.

lpfnCondition The procedure instance address of the optional, application-supplied condition function which will make an accept/reject decision based on the caller information passed in as parameters, and optionally create and/or join a socket group by assigning an appropriate value to the result parameter *g* of this function.

dwCallbackData The callback data passed back to the application as the value of the *dwCallbackData* parameter of the condition function. This parameter is not interpreted by WinSock.

Remarks This routine extracts the first connection on the queue of pending connections on *s*, and checks it against the condition function, provided the condition function is specified (i.e., not NULL). If the condition function returns CF_ACCEPT, this routine creates a new socket and performs any socket grouping as indicated by the result parameter *g* in the condition function . The newly created socket has the same properties as *s* including asynchronous events registered with **WSAAsyncSelect()** or with **WSAEventSelect()**, but **not** including the listening socket's group ID, if any. If the condition function returns CF_REJECT, this routine rejects the connection request. The condition function runs in the same thread as this routine does, and should return as soon as possible. If the decision cannot be made immediately, the condition function should return CF_DEFER to indicate that no decision has been made, and no action about this connection request should be taken by the service provider. When the application is ready to take action on the connection request, it will invoke **WSAAccept()** again and return either CF_ACCEPT or CF_REJECT as a return value from the condition function.

For sockets which remain in the (default) blocking mode, if no pending connections are present on the queue, **WSAAccept()** blocks the caller until a connection is present. For sockets in a non-blocking mode, if this function is called when no pending connections are present on the queue, **WSAAccept()** fails with the error WSAEWOULDBLOCK, as described below.

After **WSAAccept()** succeeds, and returns a new socket handle, that accepted socket may not be used to accept more connections. The original socket remains open, and listening for new connection requests..

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the address family in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*. On return, it will contain the actual length (in bytes) of the address returned. This call is used with connection-oriented socket types such as SOCK_STREAM. If *addr* and/or *addrlen* are equal to NULL, then no information about the remote address of the accepted socket is returned. Otherwise, these two parameters will be filled in regardless of whether the condition function is specified or what it returns.

The prototype of the condition function is as follows:

```
int CALLBACK
ConditionFunc(
        IN          LPWSABUF        lpCallerId,
        IN          LPWSABUF        lpCallerData,
        IN OUT      LPQOS           lpSQOS,
        IN OUT      LPQOS           lpGQOS,
        IN          LPWSABUF        lpCalleeId,
        OUT         LPWSABUF        lpCalleeData,
        OUT         GROUP FAR *     g,
        IN          DWORD           dwCallbackData
);
```

ConditionFunc is a placeholder for the application-supplied function name. The actual condition function must reside in a DLL or application module and be exported in the module definition file. You must use **MakeProcInstance()** to get a procedure-instance address for the callback function.

The *lpCallerId* and *lpCallerData* are value parameters which contain the address of the connecting entity and any user data that was sent along with the connection request, respectively. If no caller ID or caller data is available, the corresponding parameters will be NULL. (Many network protocols do not support connect-time caller data. Most conventional network protocols can be expected to support caller ID information at connection-request time.) The “buf” part of the WSABUF pointed to by *lpCallerId* points to a SOCKADDR. The SOCKADDR is interpreted according to its address family (typically by casting the SOCKADDR to some type specific to the address family).

lpSQOS references the flow specs for socket *s* specified by the caller, one for each direction, followed by any additional provider-specific parameters. The sending or

receiving flow spec values will be ignored as appropriate for any unidirectional sockets. A NULL value for *lpSQOS* indicates that there is no caller supplied QOS and that no negotiation is possible. A non-NULL *lpSQOS* pointer indicates that a QOS negotiation is to occur or that the provider is prepared to accept the QOS request without negotiation.

Reserved for future use with socket groups: *lpGQOS* references the flow specs for the socket group the caller is to create, one for each direction, followed by any additional provider-specific parameters. A NULL value for *lpGQOS* indicates no caller-supplied group QOS. QOS information may be returned if a QOS negotiation is to occur.

The *lpCalleeId* is a value parameter which contains the local address of the connected entity. The "buf" part of the WSABUF pointed to by *lpCalleeId* points to a SOCKADDR. The SOCKADDR is interpreted according to its address family (typically by casting the SOCKADDR to some type specific to the address family).

The *lpCalleeData* is a result parameter used by the condition function to supply user data back to the connecting entity. *lpCalleeData->len* initially contains the length of the buffer allocated by the service provider and pointed to by *lpCalleeData->buf*. A value of zero means passing user data back to the caller is not supported. The condition function should copy up to *lpCalleeData->len* bytes of data into *lpCalleeData->buf* , and then update *lpCalleeData->len* to indicate the actual number of bytes transferred. If no user data is to be passed back to the caller, the condition function should set *lpCalleeData->len* to zero. The format of all address and user data is specific to the address family to which the socket belongs.

Reserved for future use with socket groups: The result parameter *g* is assigned within the condition function to indicate the following actions:

if &*g* is an existing socket group ID, add *s* to this group, provided all the requirements set by this group are met; or

if &*g* = SG_UNCONSTRAINED_GROUP, create an unconstrained socket group and have *s* as the first member; or

if &*g* = SG_CONSTRAINED_GROUP, create a constrained socket group and have *s* as the first member; or

if &*g* = zero, no group operation is performed.

For unconstrained groups, any set of sockets may be grouped together as long as they are supported by a single service provider. A constrained socket group may consist only of connection-oriented sockets, and requires that connections on all grouped sockets be to the same address on the same host. For newly created socket groups, the new group ID can be retrieved by using **getsockopt()** with option SO_GROUP_ID, if this operation completes successfully. A socket group and its associated ID remain valid until the last socket belonging to this socket group is closed. Socket group IDs are unique across all processes for a given service provider.

The *dwCallbackData* parameter value passed to the condition function is the value passed as the *dwCallbackData* parameter in the original **WSAAccept()** call. This value is interpreted only by the WinSock 2 client. This allows a client to pass some context information from the **WSAAccept()** call site through to the condition function. This gives the condition function any additional information required to determine whether to accept the connection or not. A typical usage is to pass a (suitably cast) pointer to a data structure containing references to application-defined objects with which this socket is associated.

Return Value If no error occurs, **WSAAccept()** returns a value of type SOCKET which is a descriptor for the accepted socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

The integer referred to by *addrlen* initially contains the amount of space pointed to by *addr*. On return it will contain the actual length in bytes of the address returned.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAECONNREFUSED	The connection request was forcefully rejected as indicated in the return value of the condition function (CF_REJECT).
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *addrlen* argument is too small, or *addr* or *lpfnCondition* are not part of the user address space.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress.
WSAEINVAL	**listen()** was not invoked prior to **WSAAccept()**, parameter *g* specified in the condition function is not a valid value, the source address of the incoming connection request is not consistent with that of the constrained group the parameter *g* is referring to, the return value of the condition function is not a valid one, or any case where the specified socket is in an invalid state.
WSAEMFILE	The queue is non-empty upon entry to **WSAAccept()** and there are no socket descriptors available.
WSAENOBUFS	No buffer space is available.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The referenced socket is not a type that supports connection-oriented service.
WSATRY_AGAIN	The acceptance of the connection request was deferred as indicated in the return value of the condition function (CF_DEFER).
WSAEWOULDBLOCK	The socket is marked as non-blocking and no connections are present to be accepted.
WSAEACCES	The connection request that was offered has timed out or been withdrawn.

See Also **accept(), bind(), connect(), getsockopt(), listen(), select(), socket(), WSAAsyncSelect(), WSAConnect().**

4.23. WSAAsyncSelect()

Description Request Windows message-based notification of network events for a socket.

```
#include < winsock2.h >

int WSAAPI
WSAAsyncSelect (
        IN      SOCKET          s,
        IN      HWND            hWnd,
        IN      unsigned int    wMsg,
        IN      long            lEvent
);
```

s	A descriptor identifying the socket for which event notification is required.
hWnd	A handle identifying the window which should receive a message when a network event occurs.
wMsg	The message to be received when a network event occurs.
lEvent	A bitmask which specifies a combination of network events in which the application is interested.

Remarks This function is used to request that the WinSock DLL should send a message to the window *hWnd* whenever it detects any of the network events specified by the *lEvent* parameter. The message which should be sent is specified by the *wMsg* parameter. The socket for which notification is required is identified by *s*.

This function automatically sets socket *s* to non-blocking mode, regardless of the value of *lEvent*. See **ioctlsocket()** about how to set the non-blocking socket back to blocking mode.

The *lEvent* parameter is constructed by or'ing any of the values specified in the following list.

Value	Meaning
FD_READ	Want to receive notification of readiness for reading
FD_WRITE	Want to receive notification of readiness for writing
FD_OOB	Want to receive notification of the arrival of out-of-band data
FD_ACCEPT	Want to receive notification of incoming connections
FD_CONNECT	Want to receive notification of completed connection or multipoint “join” operation
FD_CLOSE	Want to receive notification of socket closure
FD_QOS	Want to receive notification of socket Quality of Service (QOS) changes
FD_GROUP_QOS	Reserved for future use with socket groups: Want to receive notification of socket group Quality of Service (QOS) changes
FD_ROUTING_INTERFACE_CHANGE	

Want to receive notification of routing interface changes for the specified destination(s)

FD_ADDRESS_LIST_CHANGE
Want to receive notification of local address list changes for the socket's protocol family

Issuing a **WSAAsyncSelect()** for a socket cancels any previous **WSAAsyncSelect()** or **WSAEventSelect()** for the same socket. For example, to receive notification for both reading and writing, the application must call **WSAAsyncSelect()** with both FD_READ and FD_WRITE, as follows:

```
rc = WSAAsyncSelect(s, hWnd, wMsg, FD_READ|FD_WRITE);
```

It is not possible to specify different messages for different events. The following code will not work; the second call will cancel the effects of the first, and only FD_WRITE events will be reported with message wMsg2:

```
rc = WSAAsyncSelect(s, hWnd, wMsg1, FD_READ);
rc = WSAAsyncSelect(s, hWnd, wMsg2, FD_WRITE);
```

To cancel all notification – i.e., to indicate that WinSock should send no further messages related to network events on the socket – *lEvent* should be set to zero.

```
rc = WSAAsyncSelect(s, hWnd, 0, 0);
```

Although in this instance **WSAAsyncSelect()** immediately disables event message posting for the socket, it is possible that messages may be waiting in the application's message queue. The application must therefore be prepared to receive network event messages even after cancellation. Closing a socket with **closesocket()** also cancels **WSAAsyncSelect()** message sending, but the same caveat about messages in the queue prior to the **closesocket()** still applies.

Since an **accept()**'ed socket has the same properties as the listening socket used to accept it, any **WSAAsyncSelect()** events set for the listening socket apply to the accepted socket. For example, if a listening socket has **WSAAsyncSelect()** events FD_ACCEPT, FD_READ, and FD_WRITE, then any socket accepted on that listening socket will also have FD_ACCEPT, FD_READ, and FD_WRITE events with the same *wMsg* value used for messages. If a different *wMsg* or events are desired, the application should call **WSAAsyncSelect()**, passing the accepted socket and the desired new information.[2]

When one of the nominated network events occurs on the specified socket *s*, the application's window *hWnd* receives message *wMsg*. The *wParam* argument identifies the socket on which a network event has occurred. The low word of *lParam* specifies the network event that has occurred. The high word of *lParam* contains any error code. The error code be any error as defined in **winsock2.h**. Note: Upon receipt of an event

[2]Note that there is a timing window between the **accept()** call and the call to **WSAAsyncSelect()** to change the events or *wMsg*. An application which desires a different *wMsg* for the listening and **accept()**'ed sockets should ask for only FD_ACCEPT events on the listening socket, then set appropriate events after the **accept()**. Since FD_ACCEPT is never sent for a connected socket and FD_READ, FD_WRITE, FD_OOB, and FD_CLOSE are never sent for listening sockets, this will not impose difficulties.

notification message the **WSAGetLastError()** function cannot be used to check the error value, because the error value returned may differ from the value in the high word of *lParam*.

The error and event codes may be extracted from the *lParam* using the macros WSAGETSELECTERROR and WSAGETSELECTEVENT, defined in **winsock2.h** as:

```
#define WSAGETSELECTERROR(lParam)             HIWORD(lParam)
#define WSAGETSELECTEVENT(lParam)             LOWORD(lParam)
```

The use of these macros will maximize the portability of the source code for the application.

The possible network event codes which may be returned are as follows:

Value	Meaning
FD_READ	Socket *s* ready for reading
FD_WRITE	Socket *s* ready for writing
FD_OOB	Out-of-band data ready for reading on socket s
FD_ACCEPT	Socket *s* ready for accepting a new incoming connection
FD_CONNECT	Connection or multipoint "join" operation initiated on socket *s* completed
FD_CLOSE	Connection identified by socket *s* has been closed
FD_QOS	Quality of Service associated with socket *s* has changed
FD_GROUP_QOS	Reserved for future use with socket groups: Quality of Service associated with the socket group to which *s* belongs has changed
FD_ROUTING_INTERFACE_CHANGE	Local interface that should be used to send to the specified destination has changed
FD_ADDRESS_LIST_CHANGE	The list of addresses of the socket's protocol family to which the application client can bind has changed

Return Value The return value is 0 if the application's declaration of interest in the network event set was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Comments Although **WSAAsyncSelect()** can be called with interest in multiple events, the application window will receive a single message for each network event.

As in the case of the **select()** function, **WSAAsyncSelect()** will frequently be used to determine when a data transfer operation (**send()** or **recv()**) can be issued with the expectation of immediate success. Nevertheless, a robust application must be prepared for the possibility that it may receive a message and issue a Winsock 2 call which returns WSAEWOULDBLOCK immediately. For example, the following sequence of events is possible:

(i) data arrives on socket **s**; Winsock 2 posts **WSAAsyncSelect** message
(ii) application processes some other message
(iii) while processing, application issues an **ioctlsocket(s, FIONREAD...)** and notices that there is data ready to be read
(iv) application issues a **recv(s,...)** to read the data

(v) application loops to process next message, eventually reaching the **WSAAsyncSelect** message indicating that data is ready to read

(vi) application issues **recv(s,...)**, which fails with the error WSAEWOULDBLOCK.

Other sequences are possible.

The Winsock DLL will not continually flood an application with messages for a particular network event. Having successfully posted notification of a particular event to an application window, no further message(s) for that network event will be posted to the application window until the application makes the function call which implicitly reenables notification of that network event.

Event	Re-enabling function
FD_READ	**recv()**, **recvfrom(), WSARecv()**, or **WSARecvFrom()**
FD_WRITE	**send()**, **sendto(), WSASend()**, or **WSASendTo()**
FD_OOB	**recv()**, **recvfrom(), WSARecv()**, or **WSARecvFrom()**
FD_ACCEPT	**accept()** or **WSAAccept()** unless the error code is WSATRY_AGAIN indicating that the condition function returned CF_DEFER
FD_CONNECT	NONE
FD_CLOSE	NONE
FD_QOS	**WSAIoctl()** with command SIO_GET_QOS
FD_GROUP_QOS	Reserved for future use with socket groups: **WSAIoctl()** with command SIO_GET_GROUP_QOS
FD_ROUTING_INTERFACE_CHANGE	**WSAIoctl()** with command SIO_ROUTING_INTERFACE_CHANGE
FD_ADDRESS_LIST_CHANGE	**WSAIoctl()** with command SIO_ADDRESS_LIST_CHANGE

Any call to the reenabling routine, even one which fails, results in reenabling of message posting for the relevant event.

For FD_READ, FD_OOB, and FD_ACCEPT events, message posting is "level-triggered." This means that if the reenabling routine is called and the relevant condition is still met after the call, a **WSAAsyncSelect()** message is posted to the application. This allows an application to be event-driven and not be concerned with the amount of data that arrives at any one time. Consider the following sequence:

(i) network transport stack receives 100 bytes of data on socket **s** and causes Winsock 2 to post an FD_READ message.

(ii) The application issues **recv(s, buffptr, 50, 0)** to read 50 bytes.

(iii) another FD_READ message is posted since there is still data to be read.

With these semantics, an application need not read all available data in response to an FD_READ message--a single **recv()** in response to each FD_READ message is appropriate. If an application issues multiple **recv()** calls in response to a single FD_READ, it may receive multiple FD_READ messages. Such an application may wish to disable FD_READ messages before starting the **recv()** calls by calling **WSAAsyncSelect()** with the FD_READ event not set.

The FD_QOS and FD_GROUP_QOS events are considered “edge triggered.” A message will be posted exactly once when a QOS change occurs. Further messages will not be forthcoming until either the provider detects a further change in QOS or the application renegotiates the QOS for the socket.

The FD_ROUTING_INTERFACE_CHANGE message is posted when the local interface that should be used to reach the destination specified in WSAIoctl() with SIO_ROUTING_INTERFACE_CHANGE changes **AFTER** such IOCTL has been issued.

The FD_ADDRESS_LIST_CHANGE message is posted when the list of addresses to which the application can bind changes **AFTER** WSAIoctl() with SIO_ADDRESS_LIST_CHANGE has been issued.

If any event has already happened when the application calls **WSAAsyncSelect()** or when the reenabling function is called, then a message is posted as appropriate. For example, consider the following sequence: 1) an application calls **listen()**, 2) a connect request is received but not yet accepted, 3) the application calls **WSAAsyncSelect()** specifying that it wants to receive FD_ACCEPT messages for the socket. Due to the persistence of events, Winsock 2 posts an FD_ACCEPT message immediately.

The FD_WRITE event is handled slightly differently. An FD_WRITE message is posted when a socket is first connected with **connect()/WSAConnect()** (after FD_CONNECT, if also registered) or accepted with **accept()/WSAAccept()**, and then after a send operation fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, an application can assume that sends are possible starting from the first FD_WRITE message and lasting until a send returns WSAEWOULDBLOCK. After such a failure the application will be notified that sends are again possible with an FD_WRITE message.

The FD_OOB event is used only when a socket is configured to receive out-of-band data separately (see section *3.5. Out-Of-Band data* for a discussion of this topic). If the socket is configured to receive out-of-band data in-line, the out-of-band (expedited) data is treated as normal data and the application should register an interest in, and will receive, FD_READ events, not FD_OOB events. An application may set or inspect the way in which out-of-band data is to be handled by using **setsockopt()** or **getsockopt()** for the SO_OOBINLINE option.

The error code in an FD_CLOSE message indicates whether the socket close was graceful or abortive. If the error code is 0, then the close was graceful; if the error code is WSAECONNRESET, then the socket's virtual circuit was reset. This only applies to connection-oriented sockets such as SOCK_STREAM.

The FD_CLOSE message is posted when a close indication is received for the virtual circuit corresponding to the socket. In TCP terms, this means that the FD_CLOSE is posted when the connection goes into the TIME WAIT or CLOSE WAIT states. This results from the remote end performing a **shutdown()** on the send side or a **closesocket().** FD_CLOSE should only be posted after all data is read from a socket, but an application should check for remaining data upon receipt of FD_CLOSE to avoid any possibility of losing data.

Please note your application will receive ONLY an FD_CLOSE message to indicate closure of a virtual circuit, and only when all the received data has been read if this is a graceful close. It will NOT receive an FD_READ message to indicate this condition.

The FD_QOS or FD_GROUP_QOS message is posted when any field in the flow spec associated with socket *s* or the socket group that *s* belongs to has changed, respectively. Applications should use **WSAIoctl()** with command SIO_GET_QOS or SIO_GET_GROUP_QOS to get the current QOS for socket *s* or for the socket group *s* belongs to, respectively.

The FD_ROUTING_INTERFACE_CHANGE and FD_ADDRESS_LIST_CHANGE events are considered “edge triggered” as well. A message will be posted exactly once when a change occurs after the application has request the notification by issuing **WSAIoctl()** with SIO_ROUTING_INTERFACE_CHANGE or SIO_ADDRESS_LIST_CHANGE correspondingly. Further messages will not be forthcoming until the application reissues the **IOCTL AND** another change is detected since the IOCTL has been issued.

Here is a summary of events and conditions for each asynchronous notification message:

- **FD_READ**:
  1) when **WSAAsyncSelect()** called, if there is data currently available to receive,
  2) when data arrives, if FD_READ not already posted,
  3) after **recv()** or **recvfrom()** called (with or without MSG_PEEK), if data is still available to receive.
  (Note: when **setsockopt()** SO_OOBINLINE is enabled "data" includes both normal data and out-of-band (OOB) data in the instances noted above.

- **FD_WRITE**:
  1) when **WSAAsyncSelect()** called, if a **send()** or **sendto()** is possible
  2) after **connect()** or **accept()** called, when connection established
  3) after **send()** or **sendto()** fail with WSAEWOULDBLOCK, when **send()** or **sendto()** are likely to succeed,
  4) after **bind()** on a connectionless socket. FD_WRITE may or may not occur at this time (implementation dependent). In any case, a connectionless socket is always writeable immediately after **bind()**.

- **FD_OOB**: Only valid when **setsockopt()** SO_OOBINLINE Is disabled (default).
  1) when **WSAAsyncSelect()** called, if there is OOB data currently available to receive with the MSG_OOB flag,
  2) when OOB data arrives, if FD_OOB not already posted,
  3) after **recv()** or **recvfrom()** called with *or without* MSG_OOB flag, if OOB data is still available to receive.

- **FD_ACCEPT**:
  1) when **WSAAsyncSelect()** called, if there is currently a connection request available to accept,
  2) when a connection request arrives, if FD_ACCEPT not already posted,
  3) after **accept()** called, if there is another connection request available to accept.

- **FD_CONNECT**:
  1) when **WSAAsyncSelect()** called, if there is currently a connection established,

2) after **connect()** called, when connection is established (even when **connect()** succeeds immediately, as is typical with a datagram socket, and even when it fails immediately).
3) after **WSAJoinLeaf()** called, when join operation completes.
4) after **connect()**, **WSAConnect()**, or **WSAJoinLeaf()** was called with a non-blocking, connection-oriented socket. The initial operation returned with a specific error of WSAEWOULDBLOCK, but the network operation went ahead. Whether the operation eventually succeeds or not, when the outcome has been determined, FD_CONNECT happens. The client should check the error code to determine whether the outcome was success or failure.

- **FD_CLOSE**: Only valid on connection-oriented sockets (e.g. SOCK_STREAM)
  1) when **WSAAsyncSelect()** called, if socket connection has been closed,
  2) after remote system initiated graceful close, when no data currently available to receive (note: if data has been received and is waiting to be read when the remote system initiates a graceful close, the FD_CLOSE is not delivered until all pending data has been read),
  3) after local system initiates graceful close with **shutdown()** and remote system has responded with "End of Data" notification (e.g. TCP FIN), when no data currently available to receive,
  4) when remote system aborts connection (e.g. sent TCP RST), and *lParam* will contain WSAECONNRESET error value.
  Note: FD_CLOSE is *not* posted after **closesocket()** is called.

- **FD_QOS**:
  1) when **WSAAsyncSelect()** called, if the QOS associated with the socket has been changed,
  2) after **WSAIoctl()** with SIO_GET_QOS called, when the QOS is changed.

- **FD_GROUP_QOS**:
  Reserved for future use with socket groups:
  1) when **WSAAsyncSelect()** called, if the group QOS associated with the socket has been changed,
  2) after **WSAIoctl()** with SIO_GET_GROUP_QOS called, when the group QOS is changed.

- **FD_ROUTING_INTERFACE_CHANGE**:
  1) after **WSAIoctl()** with SIO_ROUTING_INTERFACE_CHANGE called, when the local interface that should be used to reach the destination specified in the IOCTL changes.

- **FD_ADDRESS_LIST_CHANGE**:
  1) after **WSAIoctl()** with SIO_ADDRESS_LIST_CHANGE called, when the list of local addresses to which the application can bind changes.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.

WSAEINVAL	Indicates that one of the specified parameters was invalid such as the window handle not referring to an existing window, or the specified socket is in an invalid state.
WSAEINPROGRESS	A blocking Winsock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor is not a socket.

Additional error codes may be set when an application window receives a message. This error code is extracted from the *lParam* in the reply message using the WSAGETSELECTERROR macro. Possible error codes for each network event are:

Event: FD_CONNECT

Error Code	Meaning
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAENETUNREACH	The network can't be reached from this host at this time.
WSAEFAULT	The namelen argument is incorrect.
WSAEINVAL	The socket is already bound to an address.
WSAEISCONN	The socket is already connected.
WSAEMFILE	No more file descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAENOTCONN	The socket is not connected.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection.

Event: FD_CLOSE

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.
WSAECONNRESET	The connection was reset by the remote side.
WSAECONNABORTED	The connection was aborted due to timeout or other failure.

Event: FD_READ
Event: FD_WRITE
Event: FD_OOB
Event: FD_ACCEPT
Event: FD_QOS

Event: FD_GROUP_QOS

Event: FD_ADDRESS_LIST_CHANGE

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.

Event: FD_ROUTING_INTERFACE_CHANGE

Error Code	Meaning
WSAENETUNREACH	The specified destination is no longer reachable
WSAENETDOWN	The network subsystem has failed.

See Also **select(), WSAEventSelect()**

4.24. WSACancelBlockingCall()

Description Cancel a blocking call which is currently in progress. **WSACancelBlockingCall()** is only available for WinSock 1.1 apps (that is, if at least one thread within the process negotiates version 1.0 or 1.1 at **WSAStartup()**).

Important Note: *This function is for backwards compatibility with WinSock 1.1 and is not considered part of the WinSock 2 specification. WinSock 2 applications should* ***not*** *use this function.*

#include <winsock2.h>

int WSAAPI WSACancelBlockingCall (void);

Remarks This function cancels any outstanding blocking operation for this thread. It is normally used in two situations:

(1) An application is processing a message which has been received while a blocking call is in progress. In this case, **WSAIsBlocking()** will be true.

(2) A blocking call is in progress, and WinSock has called back to the application's "blocking hook" function (as established by **WSASetBlockingHook()**).

In each case, the original blocking call will terminate as soon as possible with the error WSAEINTR. (In (1), the termination will not take place until Windows message scheduling has caused control to revert to the blocking routine in WinSock. In (2), the blocking call will be terminated as soon as the blocking hook function completes.)

In the case of a blocking **connect()**operation, WinSock will terminate the blocking call as soon as possible, but it may not be possible for the socket resources to be released until the connection has completed (and then been reset) or timed out. This is likely to be noticeable only if the application immediately tries to open a new socket (if no sockets are available), or to **connect()** to the same peer.

Canceling an **accept()** or a **select()** call does not adversely impact the sockets passed to these calls. Only the particular call fails; any operation that was legal before the cancel is legal after the cancel, and the state of the socket is not affected in any way.

Canceling any operation other than **accept()** and **select()** can leave the socket in an indeterminate state. If an application cancels a blocking operation on a socket, the only operation that the application can depend on being able to perform on the socket is a call to **closesocket()**, although other operations may work on some WinSock implementations. If an application desires maximum portability, it must be careful not to depend on performing operations after a cancel. An application may reset the connection by setting the timeout on SO_LINGER to 0 and calling **closesocket()**.

If a cancel operation compromised the integrity of a SOCK_STREAM's data stream in any way, the WinSock provider will reset the connection and fail all future operations other than **closesocket()** with WSAECONNABORTED.

Return Value The value returned by **WSACancelBlockingCall()** is 0 if the operation was successfully canceled. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Comments

Note that it is possible that the network operation completes before the **WSACancelBlockingCall()** is processed, for example if data is received into the user buffer at interrupt time while the application is in a blocking hook. In this case, the blocking operation will return successfully as if **WSACancelBlockingCall()** had never been called. Note that the **WSACancelBlockingCall()** still succeeds in this case; the only way to know with certainty that an operation was actually canceled is to check for a return code of WSAEINTR from the blocking call.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that there is no outstanding blocking call.
WSAEOPNOTSUPP	The caller is not a WinSock 1.0 or 1.1 client.

4.25. WSACleanup()

Description Terminate use of the WinSock DLL.

#include <winsock2.h>

int WSAAPI WSACleanup (void);

Remarks An application or DLL is required to perform a successful **WSAStartup()** call before it can use WinSock services. When it has completed the use of WinSock, the application or DLL must call **WSACleanup()** to deregister itself from a WinSock implementation and allow the implementation to free any resources allocated on behalf of the application or DLL. Any pending blocking or asynchronous calls issued by any thread in this process are canceled without posting any notification messages, or signaling any event objects. Any pending overlapped send and receive operations (**WSASend()**/**WSASendTo()**/**WSARecv()**/**WSARecvFrom()** with an overlapped socket) issued by any thread in this process are also canceled without setting the event object or invoking the completion routine, if specified. In this case, the pending overlapped operations fail with the error status WSA_OPERATION_ABORTED. Any sockets open when **WSACleanup()** is called are reset and automatically deallocated as if **closesocket()** was called; sockets which have been closed with **closesocket()** but which still have pending data to be sent may be affected--the pending data may be lost if the WinSock DLL is unloaded from memory as the application exits. To insure that all pending data is sent an application should use **shutdown()** to close the connection, then wait until the close completes before calling **closesocket()** and **WSACleanup()**. All resources and internal state, such as queued un-posted messages, must be deallocated so as to be available to the next user.

There must be a call to **WSACleanup()** for every successful call to **WSAStartup()** made by a task. Only the final **WSACleanup()** for that task does the actual cleanup; the preceding calls simply decrement an internal reference count in the WinSock DLL.

Return Value The return value is 0 if the operation was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Comments Attempting to call **WSACleanup()** from within a blocking hook and then failing to check the return code is a common programming error in WinSock 1.1 applications. If an application needs to quit while a blocking call is outstanding, the application must first cancel the blocking call with **WSACancelBlockingCall()** then issue the **WSACleanup()** call once control has been returned to the application.

In a multithreaded environment, **WSACleanup()** terminates WinSock operations for all threads.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.

See Also **closesocket()**, **shutdown()**, **WSAStartup()**

4.26. WSACloseEvent()

Description Closes an open event object handle.

```
#include <winsock2.h>

BOOL WSAAPI
WSACloseEvent(
    IN      WSAEVENT    hEvent
);
```

hEvent Identifies an open event object handle.

Remarks The handle to the event object is closed so that further references to this handle will fail with the error WSA_INVALID_HANDLE.

Return Value If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSA_INVALID_HANDLE	*hEvent* is not a valid event object handle.

See Also **WSACreateEvent(), WSAEnumNetworkEvents(), WSAEventSelect(), WSAGetOverlappedResult(), WSARecv(), WSARecvFrom(), WSAResetEvent(), WSASend(), WSASendTo(), WSASetEvent(), WSAWaitForMultipleEvents().**

4.27. WSAConnect()

Description Establish a connection to a peer, exchange connect data, and specify needed quality of service based on the supplied flow spec.

```
#include <winsock2.h>

int WSAAPI
WSAConnect (
        IN      SOCKET                      s,
        IN      const struct sockaddr FAR * name,
        IN      int                         namelen,
        IN      LPWSABUF                    lpCallerData,
        OUT     LPWSABUF                    lpCalleeData,
        IN      LPQOS                       lpSQOS,
        IN      LPQOS                       lpGQOS
);
```

s	A descriptor identifying an unconnected socket.
name	The name of the peer to which the socket is to be connected.
namelen	The length of the *name*.
lpCallerData	A pointer to the user data that is to be transferred to the peer during connection establishment.
lpCalleeData	A pointer to the user data that is to be transferred back from the peer during connection establishment.
lpSQOS	A pointer to the flow specs for socket *s*, one for each direction.
lpGQOS	Reserved for future use with socket groups: A pointer to the flow specs for the socket group (if applicable).

Remarks This function is used to create a connection to the specified destination, and to perform a number of other ancillary operations that occur at connect time as well. If the socket, *s*, is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound.

For connection-oriented sockets (e.g., type SOCK_STREAM), an active connection is initiated to the foreign host using *name* (an address in the name space of the socket; for a detailed description, please see **bind()**). When this call completes successfully, the socket is ready to send/receive data. If the address field of the *name* structure is all zeroes, **WSAConnect()** will return the error WSAEADDRNOTAVAIL. Any attempt to re-connect an active connection will fail with the error code WSAEISCONN.

For connection-oriented, non-blocking sockets it is often not possible to complete the connection immediately. In such a case, this function returns with the error WSAEWOULDBLOCK but the operation proceeds. When the success or failure outcome becomes known, it may be reported in one of several ways depending on how the client registers for notification. If the client uses **select()** success is reported in the **writefds** set and failure is reported in the **exceptfds** set. If the client uses

WSAAsyncSelect() or **WSAEventSelect()**, the notification is announced with FD_CONNECT and the error code associated with the FD_CONNECT indicates either success or a specific reason for failure.

For a connectionless socket (e.g., type SOCK_DGRAM), the operation performed by **WSAConnect()** is merely to establish a default destination address so that the socket may be used on subsequent connection-oriented send and receive operations (**send(), WSASend(), recv(), WSARecv())**. Any datagrams received from an address other than the destination address specified will be discarded. If the address field of the *name* structure is all zeroes, the socket will be "dis-connected" - the default remote address will be indeterminate, so **send()/WSASend()** and **recv()/WSARecv()** calls will return the error code WSAENOTCONN, although **sendto()/WSASendTo()** and **recvfrom()/WSARecvFrom()** may still be used. The default destination may be changed by simply calling **WSAConnect()** again, even if the socket is already "connected". Any datagrams queued for receipt are discarded if *name* is different from the previous **WSAConnect()**.

For connectionless sockets, *name* may indicate any valid address, including a broadcast address. However, to connect to a broadcast address, a socket must have **setsockopt()** SO_BROADCAST enabled, otherwise **WSAConnect()** will fail with the error code WSAEACCES.

On connectionless sockets, exchange of user to user data is not possible and the corresponding parameters will be silently ignored.

The application is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specifies.

The *lpCallerData* is a value parameter which contains any user data that is to be sent along with the connection request. If *lpCallerData* is NULL, no user data will be passed to the peer. The *lpCalleeData* is a result parameter which will contain any user data passed back from the peer as part of the connection establishment. *lpCalleeData->len* initially contains the length of the buffer allocated by the application and pointed to by *lpCalleeData->buf*. *lpCalleeData->len* will be set to 0 if no user data has been passed back. The *lpCalleeData* information will be valid when the connection operation is complete. For blocking sockets, this will be when the **WSAConnect()** function returns. For non-blocking sockets, this will be after the FD_CONNECT notification has occurred. If *lpCalleeData* is NULL, no user data will be passed back. The exact format of the user data is specific to the address family to which the socket belongs.

At connect time, an application may use the *lpSQOS* and/or *lpGQOS* parameters to override any previous QOS specification made for the socket via **WSAIoctl()** with either the SIO_SET_QOS or SIO_SET_GROUP_QOS opcodes.

lpSQOS specifies the flow specs for socket *s*, one for each direction, followed by any additional provider-specific parameters. If either the associated transport provider in general or the specific type of socket in particular cannot honor the QOS request, an error will be returned as indicated below. The sending or receiving flow spec values will be ignored, respectively, for any unidirectional sockets. If no provider-specific parameters are supplied, the *buf* and *len* fields of *lpSQOS->ProviderSpecific* should be set to NULL and 0, respectively. A NULL value for *lpSQOS* indicates no application supplied QOS.

Reserved for future use with socket groups: *lpGQOS* specifies the flow specs for the socket group (if applicable), one for each direction, followed by any additional provider-specific parameters. If no provider-specific parameters are supplied, the *buf* and *len* fields of *lpGQOS->ProviderSpecific* should be set to NULL and 0, respectively. A NULL value for *lpGQOS* indicates no application-supplied group QOS. This parameter will be ignored if *s* is not the creator of the socket group.

Comments

When connected sockets break (i.e. become closed for whatever reason), they should be discarded and recreated. It is safest to assume that when things go awry for any reason on a connected socket, the application must discard and recreate the needed sockets in order to return to a stable point.

Return Value

If no error occurs, **WSAConnect()** returns 0. Otherwise, it returns SOCKET_ERROR, and a specific error code may be retrieved by calling **WSAGetLastError().**

On a blocking socket, the return value indicates success or failure of the connection attempt.

With a non-blocking socket, the connection attempt may not be completed immediately - in this case **WSAConnect()** will return SOCKET_ERROR, and **WSAGetLastError()** will return WSAEWOULDBLOCK. In this case the application may:

1. Use **select()** to determine the completion of the connection request by checking if the socket is writeable, or

2. If your application is using **WSAAsyncSelect()** to indicate interest in connection events, then your application will receive an FD_CONNECT notification when the connect operation is complete (successfully, or not), or

3. If your application is using **WSAEventSelect()** to indicate interest in connection events, then the associated event object will be signaled when the connect operation is complete (successfully, or not).

For a non-blocking socket, until the connection attempt completes all subsequent calls to **WSAConnect()** on the same socket will fail with the error code WSAEALREADY, and WSAEISCONN when the connection completes successfully. Due to ambiguities in version 1.1 of the Windows Sockets specification, error codes returned from **connect()** while a connection is already pending may vary among implementations. As a result, it isn't recommended that applications use multiple calls to **connect()** to detect connection completion. If they do, they must be prepared to handle WSAEINVAL and WSAEWOULDBLOCK error values the same way that they handle WSAEALREADY, to assure robust execution.
If the return error code indicates the connection attempt failed (i.e. WSAECONNREFUSED, WSAENETUNREACH, WSAETIMEDOUT) the application may call **WSAConnect()** again for the same socket.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.

WSAEADDRINUSE	The socket's local address is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs at the time of **bind()**, but could be delayed until this function if the **bind()** was to a partially wild-card address (involving ADDR_ANY) and if a specific address needs to be "committed" at the time of this function.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	A non-blocking **connect()/WSAConnect()** call is in progress on the specified socket.
WSAEADDRNOTAVAIL	The remote address is not a valid address (e.g., ADDR_ANY).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was rejected.
WSAEFAULT	The *name* or the *namelen* argument is not a valid part of the user address space, the *namelen* argument is too small, the buffer length for *lpCalleeData*, *lpSQOS*, and *lpGQOS* are too small, or the buffer length for *lpCallerData* is too large.
WSAEINVAL	The parameter *s* is a listening socket, or the destination address specified is not consistent with that of the constrained group the socket belongs to.
WSAEISCONN	The socket is already connected (connection-oriented sockets only).
WSAENETUNREACH	The network can't be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The flow specs specified in *lpSQOS* and *lpGQOS* cannot be satisfied.
WSAEPROTONOSUPPORT	The *lpCallerData* argument is not supported by the service provider.

WSAETIMEDOUT	Attempt to connect timed out without establishing a connection.
WSAEWOULDBLOCK	The socket is marked as non-blocking and the connection cannot be completed immediately.
WSAEACCES	Attempt to connect datagram socket to broadcast address failed because **setsockopt()** SO_BROADCAST is not enabled.

See Also **accept(), bind(), connect(), getsockname(), getsockopt(), socket(), select(), WSAAsyncSelect(), WSAEventSelect().**

4.28. WSACreateEvent()

Description Creates a new event object.

#include <winsock2.h>

WSAEVENT WSAAPI WSACreateEvent(void);

Remarks The event object created by this function is manual reset, with an initial state of nonsignaled. WinSock 2 event objects are system objects in Win32 environments. Therefore, if a Win32 application desires auto reset events, it may call the native **CreateEvent()** Win32 API directly. The scope of an event object is limited to the process in which it is created.

Return Value If the function succeeds, the return value is the handle of the event object.

If the function fails, the return value is WSA_INVALID_EVENT. To get extended error information, call **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to create the event object.

See Also **WSACloseEvent()**, **WSAEnumNetworkEvents(), WSAEventSelect(), WSAGetOverlappedResult(), WSARecv(), WSARecvFrom(), WSAResetEvent(), WSASend(), WSASendTo(), WSASetEvent(), WSAWaitForMultipleEvents().**

4.29. WSADuplicateSocket()

Description Return a WSAPROTOCOL_INFO structure that can be used to create a new socket descriptor for a shared socket.

```
#include <winsock2.h>

int WSAAPI
WSADuplicateSocket (
        IN      SOCKET                  s,
        IN      DWORD                   dwProcessId,
        OUT     LPWSAPROTOCOL_INFO      lpProtocolInfo
);
```

s Specifies the local socket descriptor.

dwProcessId Specifies the ID of the target process for which the shared socket will be used.

lpProtocolInfo A pointer to a buffer allocated by the client that is large enough to contain a WSAPROTOCOL_INFO struct. The service provider copies the protocol info struct contents to this buffer.

Remarks This function is used to enable socket sharing between processes. A source process calls **WSADuplicateSocket()** to obtain a special WSAPROTOCOL_INFO structure. It uses some interprocess communications (IPC) mechanism to pass the contents of this structure to a target process, which in turn uses it in a call to **WSASocket()** to obtain a descriptor for the duplicated socket. Note that the special WSAPROTOCOL_INFO structure may only be used once by the target process.

Note that sockets may be shared amongst threads in a given process without using the **WSADuplicateSocket()** function, since a socket descriptor is valid in all of a process' threads.
One possible scenario for establishing and using a shared socket in a handoff mode is illustrated below:

Source Process	IPC	Destination Process
1) **WSASocket()**, **WSAConnect()**		
2) Request target process ID	⇒	
		3) Receive process ID request and respond
4) Receive process ID	⇐	
5) Call **WSADuplicateSocket()** to get a special WSAPROTOCOL_INFO structure		
6) Send WSAPROTOCOL_INFO structure to target		
	⇒	7) Receive WSAPROTOCOL_INFO structure

		8) Call **WSASocket()** to create shared socket descriptor.
10) **closesocket()**		9)Use shared socket for data exchange

Return Value If no error occurs, **WSADuplicateSocket()** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError().**

Comments The descriptors that reference a shared socket may be used independently as far as I/O is concerned. However, the WinSock interface does not implement any type of access control, so it is up to the processes involved to coordinate their operations on a shared socket. A typical use for shared sockets is to have one process that is responsible for creating sockets and establishing connections, hand off sockets to other processes which are responsible for information exchange.

Since what is duplicated are the socket descriptors and not the underlying socket, all of the state associated with a socket is held in common across all the descriptors. For example a **setsockopt()** operation performed using one descriptor is subsequently visible using a **getsockopt()** from any or all descriptors. A process may call **closesocket()** on a duplicated socket and the descriptor will become deallocated. The underlying socket, however, will remain open until **closesocket()** is called by the last remaining descriptor.

Notification on shared sockets is subject to the usual constraints of **WSAAsyncSelect()** and **WSAEventSelect()**. Issuing either of these calls using any of the shared descriptors cancels any previous event registration for the socket, regardless of which descriptor was used to make that registration. Thus, for example, a shared socket cannot deliver FD_READ events to process A and FD_WRITE events to process B. For situations when such tight coordination is required, it is suggested that developers use threads instead of separate processes.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAENOTSOCK	The descriptor is not a socket.

WSAEFAULT	The *lpProtocolInfo* argument is not a valid part of the user address space.

See Also **WSASocket()**

4.30. WSAEnumNetworkEvents()

Description Discover occurrences of network events for the indicated socket, clear internal network events record, and (optionally) reset event object.

#include <winsock2.h>

int WSAAPI
WSAEnumNetworkEvents (

	IN	**SOCKET**	*s*,
	IN	**WSAEVENT**	*hEventObject*,
	OUT	**LPWSANETWORKEVENTS**	*lpNetworkEvents*

);

s A descriptor identifying the socket.

hEventObject An optional handle identifying an associated event object to be reset.

lpNetworkEvents A pointer to a WSANETWORKEVENTS struct which is filled with a record of occurred network events and any associated error codes.

Remarks This function is used to discover which network events have occurred for the indicated socket since the last invocation of this function. It is intended for use in conjunction with **WSAEventSelect(),** which associates an event object with one or more network events. Recording of network events commences when **WSAEventSelect()** is called with a non-zero *lNetworkEvents* parameter and remains in effect until another call is made to **WSAEventSelect()** with the *lNetworkEvents* parameter set to zero, or until a call is made to **WSAAsyncSelect()**.

WSAEnumNetworkEvents() only reports network activity and errors nominated through **WSAEventSelect()**. See the descriptions of **select()** and **WSAAsyncSelect()** to find out how those functions report network activity and errors.

The socket's internal record of network events is copied to the structure referenced by *lpNetworkEvents*, whereafter the internal network events record is cleared. If *hEventObject* is non-null, the indicated event object is also reset. The WinSock provider guarantees that the operations of copying the network event record, clearing it and resetting any associated event object are atomic, such that the next occurrence of a nominated network event will cause the event object to become set. In the case of this function returning SOCKET_ERROR, the associated event object is not reset and the record of network events is not cleared.

The WSANETWORKEVENTS structure is defined as follows:

```
typedef struct _WSANETWORKEVENTS {
       long lNetworkEvents;
       int iErrorCode [FD_MAX_EVENTS];
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS;
```

The *lNetworkEvents* field of the structure indicates which of the FD_XXX network events have occurred. The *iErrorCode* array is used to contain any associated error codes, with array index corresponding to the position of event bits in lNetworkEvents. The identifiers FD_READ_BIT, FD_WRITE_BIT, etc. may be used to index the

iErrorCode array. Note that only those elements of the *iErrorCode* array are set that correspond to the bits set in *lNetworkEvents* field. Other fields are not modified (this is important for backwards compatibility with the applications that are not aware of new FD_ROUTING_INTERFACE_CHANGE and FD_ADDRESS_LIST_CHANGE events).

The following error codes may be returned along with the respective network event:

Event: FD_CONNECT

Error Code	Meaning
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAENETUNREACH	The network can't be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be connected.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection

Event: FD_CLOSE

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.
WSAECONNRESET	The connection was reset by the remote side.
WSAECONNABORTED	The connection was aborted due to timeout or other failure.

Event: FD_READ
Event: FD_WRITE
Event: FD_OOB
Event: FD_ACCEPT
Event: FD_QOS
Event: FD_GROUP_QOS
Event: FD_ADDRESS_LIST_CHANGE

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.

Event: FD_ROUTING_INTERFACE_CHANGE

Error Code	Meaning
WSAENETUNREACH	The specified destination is no longer reachable
WSAENETDOWN	The network subsystem has failed.

Return Value

The return value is 0 if the operation was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lpNetworkEvents* argument is not a valid part of the user address space.

See Also **WSAEventSelect()**

4.31. WSAEnumProtocols()

Description Retrieve information about available transport protocols.

```
#include <winsock2.h>

int WSAAPI
WSAEnumProtocols (
        IN          LPINT                   lpiProtocols,
        OUT         LPWSAPROTOCOL_INFO      lpProtocolBuffer,
        IN OUT      LPDWORD                 lpdwBufferLength
);
```

lpiProtocols A NULL-terminated array of iProtocol values. This parameter is optional; if *lpiProtocols* is NULL, information on all available protocols is returned, otherwise information is retrieved only for those protocols listed in the array.

lpProtocolBuffer A buffer which is filled with WSAPROTOCOL_INFO structures. See below for a detailed description of the contents of the WSAPROTOCOL_INFO structure.

lpdwBufferLength On input, the count of bytes in the *lpProtocolBuffer* buffer passed to **WSAEnumProtocols().** On output, the minimum buffer size that can be passed to **WSAEnumProtocols()** to retrieve all the requested information. This routine has no ability to enumerate over multiple calls; the passed-in buffer must be large enough to hold all entries in order for the routine to succeed. This reduces the complexity of the API and should not pose a problem because the number of protocols loaded on a machine is typically small.

Remarks This function is used to discover information about the collection of transport protocols and protocol chains installed on the local machine. Since layered protocols are only usable by applications when installed in protocol chains, information on layered protocols is not included in *lpProtocolBuffer*. The *lpiProtocols* parameter can be used as a filter to constrain the amount of information provided. Normally it will be supplied as a NULL pointer which will cause the routine to return information on all available transport protocols and protocol chains.

A WSAPROTOCOL_INFO struct is provided in the buffer pointed to by *lpProtocolBuffer* for each requested protocol. If the supplied buffer is not large enough (as indicated by the input value of *lpdwBufferLength*), the value pointed to by *lpdwBufferLength* will be updated to indicate the required buffer size. The application should then obtain a large enough buffer and call this function again.

The order in which the WSAPROTOCOL_INFO structs appear in the buffer coincides with the order in which the protocol entries were registered by the service provider with the WinSock DLL, or with any subsequent re-ordering that may have occurred via the WinSock applet or DLL supplied for establishing default TCP/IP providers.

Definitions **WSAPROTOCOL_INFO Structure:**

DWORD *dwServiceFlags1* - a bitmask describing the services provided by the protocol. The following values are possible:

XP1_CONNECTIONLESS -the protocol provides connectionless (datagram) service. If not set, the protocol supports connection-oriented data transfer.

XP1_GUARANTEED_DELIVERY - the protocol guarantees that all data sent will reach the intended destination.

XP1_GUARANTEED_ORDER - the protocol guarantees that data will only arrive in the order in which it was sent and that it will not be duplicated. This characteristic does not necessarily mean that the data will always be delivered, but that any data that is delivered is delivered in the order in which it was sent.

XP1_MESSAGE_ORIENTED - the protocol honors message boundaries, as opposed to a stream-oriented protocol where there is no concept of message boundaries.

XP1_PSEUDO_STREAM - this is a message oriented protocol, but message boundaries will be ignored for all receives. This is convenient when an application does not desire message framing to be done by the protocol.

XP1_GRACEFUL_CLOSE - the protocol supports two-phase (graceful) close. If not set, only abortive closes are performed.

XP1_EXPEDITED_DATA - the protocol supports expedited (urgent) data.

XP1_CONNECT_DATA - the protocol supports connect data.

XP1_DISCONNECT_DATA - the protocol supports disconnect data.

XP1_SUPPORT_BROADCAST - the protocol supports a broadcast mechanism.

XP1_SUPPORT_MULTIPOINT - the protocol supports a multipoint or multicast mechanism. Control and data plane attributes are indicated below. Refer to Appendix B. Multipoint and Multicast Semantics for additional information.

XP1_MULTIPOINT_CONTROL_PLANE - indicates whether the control plane is rooted (value = 1) or non-rooted (value = 0).

XP1_MULTIPOINT_DATA_PLANE - indicates whether the data plane is rooted (value = 1) or non-rooted (value = 0).

XP1_QOS_SUPPORTED - the protocol supports quality of service requests.

XP1_RESERVED - This bit is reserved.

XP1_UNI_SEND - the protocol is unidirectional in the send direction.

XP1_UNI_RECV - the protocol is unidirectional in the recv direction.

XP1_IFS_HANDLES - the socket descriptors returned by the provider are operating system Installable File System (IFS) handles.

XP1_PARTIAL_MESSAGE - the MSG_PARTIAL flag is supported in **WSASend()** and **WSASendTo()**.

Note that only one of XP1_UNI_SEND or XP1_UNI_RECV may be set. If a protocol can be unidirectional in either direction, two WSAPROTOCOL_INFO structs should be used. When neither bit is set, the protocol is considered to be bi-directional.

DWORD *dwServiceFlags2* - reserved for additional protocol attribute definitions.

DWORD *dwServiceFlags3*- reserved for additional protocol attribute definitions.

DWORD *dwServiceFlags4* - reserved for additional protocol attribute definitions.

DWORD *dwProviderFlags* - provide information about how this protocol is represented in the protocol catalog. The following flag values are possible:

PFL_MULTIPLE_PROTOCOL_ENTRIES - indicates that this is one of two or more entries for a single protocol (from a given provider) which is capable of implementing multiple behaviors. An example of this is SPX which, on the receiving side, can behave either as a message oriented or a stream oriented protocol.

PFL_RECOMMENDED_PROTO_ENTRY - indicates that this is the recommended or most frequently used entry for a protocol which is capable of implementing multiple behaviors.

PFL_HIDDEN - set by a provider to indicate to the WinSock 2 DLL that this protocol should not be returned in the result buffer generated by **WSAEnumProtocols()**. Obviously, a WinSock 2 application should never see an entry with this bit set.

PFL_MATCHES_PROTOCOL_ZERO - indicates that a value of zero in the *protocol* parameter of **socket()** or **WSASocket()** matches this protocol entry.

GUID *ProviderId*- A globally unique identifier assigned to the provider by the service provider vendor. This value is useful for instances where more than one service provider is able to implement a particular protocol. An application may use the *ProviderId* value to distinguish between providers that might otherwise be indistinguishable.

DWORD *dwCatalogEntryId* - A unique identifier assigned by the WinSock 2 DLL for each WSAPROTOCOL_INFO structure.

WSAPROTOCOLCHAIN *ProtocolChain* - A structure containing a counted list of Catalog Entry IDs which comprise a protocol chain. This structure is defined as follows:

```
typedef struct {
  int ChainLen;
  DWORD ChainEntries[MAX_PROTOCOL_CHAIN];
} WSAPROTOCOLCHAIN
```

If the length of the chain is 0, this WSAPROTOCOL_INFO entry represents a layered protocol which has WinSock 2 SPI as both its top and bottom edges. If the length of the

chain equals 1, this entry represents a base protocol whose Catalog Entry ID is in the *dwCatalogEntryId* field above. If the length of the chain is larger than 1, this entry represents a protocol chain which consists of one or more layered protocols on top of a base protocol. The corresponding Catalog Entry IDs are in the *ProtocolChain.ChainEntries* array starting with the layered protocol at the top (the zeroth element in the *ProtocolChain.ChainEntries* array) and ending with the base protocol. Refer to the ***WinSock 2 Service Provider Interface*** specification for more information on protocol chains.

int *iVersion* - Protocol version identifier.

int *iAddressFamily* - the value to pass as the address family parameter to the **socket()**/**WSASocket()** function in order to open a socket for this protocol. This value also uniquely defines the structure of protocol addresses (SOCKADDRs) used by the protocol.

int *iMaxSockAddr* - The maximum address length in bytes (e.g. 16 for IP version 4, the equivalent of sizeof(SOCKADDR_IN)).

int *iMinSockAddr* - The minimum address length (same as *iMaxSockAddr*, unless protocol supports variable length addressing).

int *iSocketType* - The value to pass as the socket type parameter to the **socket()** function in order to open a socket for this protocol.

int *iProtocol* - The value to pass as the protocol parameter to the **socket()** function in order to open a socket for this protocol.

int *iProtocolMaxOffset* - The maximum value that may be added to *iProtocol* when supplying a value for the *protocol* parameter to **socket()** and **WSASocket()**. Not all protocols allow a range of values. When this is the case *iProtocolMaxOffset* will be zero.

int *iNetworkByteOrder* - Currently these values are manifest constants (BIGENDIAN and LITTLEENDIAN) that indicate either “big-endian” or “little-endian” with the values 0 and 1 respectively.

int *iSecurityScheme* - Indicates the type of security scheme employed (if any). A value of SECURITY_PROTOCOL_NONE is used for protocols that do not incorporate security provisions.

DWORD *dwMessageSize* - The maximum message size supported by the protocol. This is the maximum size that can be sent from any of the host’s local interfaces. For protocols which do not support message framing, the actual maximum that can be sent to a given address may be less. There is no standard provision to determine the maximum inbound message size. The following special values are defined:

0 - the protocol is stream-oriented and hence the concept of message size is not relevant.

0x1 - the maximum outbound (send) message size is dependent on the underlying network MTU (maximum sized transmission unit) and hence cannot be known until after a socket is bound. Applications should use **getsockopt()** to retrieve the value of SO_MAX_MSG_SIZE after the socket has been bound to a local address.

0xFFFFFFFF - the protocol is message-oriented, but there is no maximum limit to the size of messages that may be transmitted.

DWORD *dwProviderReserved* - reserved for use by service providers.

char *szProtocol* - an array of characters that contains a human-readable name identifying the protocol, for example "SPX". The maximum number of characters allowed is WSAPROTOCOL_LEN, which is defined to be 255.

Return Value If no error occurs, **WSAEnumProtocols()** returns the number of protocols to be reported on. Otherwise a value of SOCKET_ERROR is returned and a specific error code may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAENOBUFS	The buffer length was too small to receive all the relevant WSAPROTOCOL_INFO structures and associated information. Pass in a buffer at least as large as the value returned in *lpdwBufferLength*.
WSAEFAULT	One or more of the *lpiProtocols, lpProtocolBuffer, or lpdwBufferLength* arguments are not a valid part of the user address space.

4.32. WSAEventSelect()

Description Specify an event object to be associated with the supplied set of FD_XXX network events.

```
#include <winsock2.h>

int WSAAPI
WSAEventSelect (
        IN      SOCKET      s,
        IN      WSAEVENT    hEventObject,
        IN      long        lNetworkEvents
);
```

s A descriptor identifying the socket.

hEventObject A handle identifying the event object to be associated with the supplied set of FD_XXX network events.

lNetworkEvents A bitmask which specifies the combination of FD_XXX network events in which the application has interest.

Remarks This function is used to specify an event object, *hEventObject*, to be associated with the selected FD_XXX network events, *lNetworkEvents*. The socket for which an event object is specified is identified by *s*. The event object is set when any of the nominated network events occur.

WSAEventSelect() operates very similarly to **WSAAsyncSelect()**, the difference being in the actions taken when a nominated network event occurs. Whereas **WSAAsyncSelect()** causes an application-specified Windows message to be posted, **WSAEventSelect()** sets the associated event object and records the occurrence of this event in an internal network event record. An application can use **WSAWaitForMultipleEvents()** to wait or poll on the event object, and use **WSAEnumNetworkEvents()** to retrieve the contents of the internal network event record and thus determine which of the nominated network events have occurred.

WSAEventSelect() is the only function that causes network activity and errors to be recorded and retrievable through **WSAEnumNetworkEvents()**. See the descriptions of **select()** and **WSAAsyncSelect()** to find out how those functions report network activity and errors.

This function automatically sets socket *s* to non-blocking mode, regardless of the value of *lNetworkEvents*. See **ioctlsocket()/WSAIoctl()** about how to set the socket back to blocking mode.

The *lNetworkEvents* parameter is constructed by or'ing any of the values specified in the following list.

Value	Meaning
FD_READ	Want to receive notification of readiness for reading
FD_WRITE	Want to receive notification of readiness for writing

FD_OOB	Want to receive notification of the arrival of out-of-band data
FD_ACCEPT	Want to receive notification of incoming connections
FD_CONNECT	Want to receive notification of completed connection or multipoint "join" operation
FD_CLOSE	Want to receive notification of socket closure
FD_QOS	Want to receive notification of socket Quality of Service (QOS) changes
FD_GROUP_QOS	Reserved for future use with socket groups: Want to receive notification of socket group Quality of Service (QOS) changes
FD_ROUTING_INTERFACE_CHANGE	Want to receive notification of routing interface changes for the specified destination
FD_ADDRESS_LIST_CHANGE	Want to receive notification of local address list changes for the socket's address family

Issuing a **WSAEventSelect()** for a socket cancels any previous **WSAAsyncSelect()** or **WSAEventSelect()** for the same socket and clears the internal network event record. For example, to associate an event object with both reading and writing network events, the application must call **WSAEventSelect()** with both FD_READ and FD_WRITE, as follows:

```
rc = WSAEventSelect(s, hEventObject, FD_READ|FD_WRITE);
```

It is not possible to specify different event objects for different network events. The following code will not work; the second call will cancel the effects of the first, and only FD_WRITE network event will be associated with hEventObject2:

```
rc = WSAEventSelect(s, hEventObject1, FD_READ);
rc = WSAEventSelect(s, hEventObject2, FD_WRITE);   //bad
```

To cancel the association and selection of network events on a socket, *lNetworkEvents* should be set to zero, in which case the *hEventObject* parameter will be ignored.

```
rc = WSAEventSelect(s, hEventObject, 0);
```

Closing a socket with **closesocket()** also cancels the association and selection of network events specified in **WSAEventSelect()** for the socket. The application, however, still must call **WSACloseEvent()** to explicitly close the event object and free any resources.

Since an **accept()**'ed socket has the same properties as the listening socket used to accept it, any **WSAEventSelect()** association and network events selection set for the listening socket apply to the accepted socket. For example, if a listening socket has **WSAEventSelect()** association of *hEventOject* with FD_ACCEPT, FD_READ, and FD_WRITE, then any socket accepted on that listening socket will also have FD_ACCEPT, FD_READ, and FD_WRITE network events associated with the same *hEventObject*. If a different *hEventObject* or network events are desired, the

application should call **WSAEventSelect()**, passing the accepted socket and the desired new information.[3]

Return Value The return value is 0 if the application's specification of the network events and the associated event object was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

As in the case of the **select()** and **WSAAsyncSelect()** functions, **WSAEventSelect()** will frequently be used to determine when a data transfer operation (**send()** or **recv()**) can be issued with the expectation of immediate success. Nevertheless, a robust application must be prepared for the possibility that the event object is set and it issues a WinSock call which returns WSAEWOULDBLOCK immediately. For example, the following sequence of operations is possible:

(i)	data arrives on socket **s**; WinSock sets the **WSAEventSelect** event object
(ii)	application does some other processing
(iii)	while processing, application issues an **ioctlsocket(s, FIONREAD...)** and notices that there is data ready to be read
(iv)	application issues a **recv(s,...)** to read the data
(v)	application eventually waits on event object specified in **WSAEventSelect()**, which returns immediately indicating that data is ready to read
(vi)	application issues **recv(s,...)**, which fails with the error WSAEWOULDBLOCK.

Other sequences are possible.

Having successfully recorded the occurrence of the network event (by setting the corresponding bit in the internal network event record) and signaled the associated event object, no further actions are taken for that network event until the application makes the function call which implicitly reenables the setting of that network event and signaling of the associated event object.

Network Event	Re-enabling function
FD_READ	**recv()**, **recvfrom(), WSARecv(),** or **WSARecvFrom()**
FD_WRITE	**send()**, **sendto(), WSASend(),** or **WSASendTo()**
FD_OOB	**recv()**, **recvfrom(), WSARecv(),** or **WSARecvFrom()**
FD_ACCEPT	**accept()** or **WSAAccept()** unless the error code returned is WSATRY_AGAIN indicating that the condition function returned CF_DEFER
FD_CONNECT	NONE
FD_CLOSE	NONE
FD_QOS	**WSAIoctl()** with command SIO_GET_QOS

[3]Note that there is a timing window between the **accept()** call and the call to **WSAEventSelect()** to change the network events or *hEventObject*. An application which desires a different *hEventObject* for the listening and **accept()**'ed sockets should ask for only FD_ACCEPT network event on the listening socket, then set appropriate network events after the **accept()**. Since FD_ACCEPT never happens to a connected socket and FD_READ, FD_WRITE, FD_OOB, and FD_CLOSE never happen to listening sockets, this will not impose difficulties.

FD_GROUP_QOS Reserved for future use with socket groups: **WSAIoctl()** with command SIO_GET_GROUP_QOS

FD_ROUTING_INTERFACE_CHANGE **WSAIoctl()** with command SIO_ROUTING_INTERFACE_CHANGE

FD_ADDRESS_LIST_CHANGE **WSAIoctl()** with command SIO_ADDRESS_LIST_CHANGE

Any call to the reenabling routine, even one which fails, results in reenabling of recording and signaling for the relevant network event and event object, respectively.

For FD_READ, FD_OOB, and FD_ACCEPT network events, network event recording and event object signaling are "level-triggered." This means that if the reenabling routine is called and the relevant network condition is still valid after the call, the network event is recorded and the associated event object is set . This allows an application to be event-driven and not be concerned with the amount of data that arrives at any one time. Consider the following sequence:

(i) transport provider receives 100 bytes of data on socket **s** and causes WinSock DLL to record the FD_READ network event and set the associated event object.

(ii) The application issues **recv(s, buffptr, 50, 0)** to read 50 bytes.

(iii) The transport provider causes WinSock DLL to record the FD_READ network event and sets the associated event object again since there is still data to be read.

With these semantics, an application need not read all available data in response to an FD_READ network event --a single **recv()** in response to each FD_READ network event is appropriate.

The FD_QOS and FD_GROUP_QOS events are considered edge triggered. A message will be posted exactly once when a QOS change occurs. Further messages will not be forthcoming until either the provider detects a further change in QOS or the application renegotiates the QOS for the socket.

The FD_ROUTING_INTERFACE_CHANGE and FD_ADDRESS_LIST_CHANGE events are considered “edge triggered” as well. A message will be posted exactly once when a change occurs **AFTER** the application has request the notification by issuing **WSAIoctl()** with SIO_ROUTING_INTERFACE_CHANGE or SIO_ADDRESS_LIST_CHANGE correspondingly. Further messages will not be forthcoming until the application reissues the IOCTL **AND** another change is detected since the IOCTL has been issued.

If a network event has already happened when the application calls **WSAEventSelect()** or when the reenabling function is called, then a network event is recorded and the associated event object is set as appropriate. For example, consider the following sequence: 1) an application calls **listen()**, 2) a connect request is received but not yet accepted, 3) the application calls **WSAEventSelect()** specifying that it is interested in the FD_ACCEPT network event for the socket. Due to the persistence of network events, WinSock records the FD_ACCEPT network event and sets the associated event object immediately.

The FD_WRITE network event is handled slightly differently. An FD_WRITE network event is recorded when a socket is first connected with **connect()/WSAConnect()** or accepted with **accept()/WSAAccept()**, and then after a send fails with WSAEWOULDBLOCK and buffer space becomes available. Therefore, an application can assume that sends are possible starting from the first FD_WRITE network event setting and lasting until a send returns WSAEWOULDBLOCK. After such a failure the application will find out that sends are again possible when an FD_WRITE network event is recorded and the associated event object is set .

The FD_OOB network event is used only when a socket is configured to receive out-of-band data separately. If the socket is configured to receive out-of-band data in-line, the out-of-band (expedited) data is treated as normal data and the application should register an interest in, and will get, FD_READ network event, not FD_OOB network event. An application may set or inspect the way in which out-of-band data is to be handled by using **setsockopt()** or **getsockopt()** for the SO_OOBINLINE option.

The error code in an FD_CLOSE network event indicates whether the socket close was graceful or abortive. If the error code is 0, then the close was graceful; if the error code is WSAECONNRESET, then the socket's virtual circuit was reset. This only applies to connection-oriented sockets such as SOCK_STREAM.

The FD_CLOSE network event is recorded when a close indication is received for the virtual circuit corresponding to the socket. In TCP terms, this means that the FD_CLOSE is recorded when the connection goes into the TIME WAIT or CLOSE WAIT states. This results from the remote end performing a **shutdown()** on the send side or a **closesocket().** FD_CLOSE should only be posted after all data is read from a socket, but an application should check for remaining data upon receipt of FD_CLOSE to avoid any possibility of losing data.

Please note WinSock will record ONLY an FD_CLOSE network event to indicate closure of a virtual circuit. It will NOT record an FD_READ network event to indicate this condition.

The FD_QOS or FD_GROUP_QOS network event is recorded when any field in the flow spec associated with socket *s* or the socket group that *s* belongs to has changed, respectively. Applications should use **WSAIoctl()** with command SIO_GET_QOS or SIO_GET_GROUP_QOS to get the current QOS for socket *s* or for the socket group *s* belongs to, respectively.

The FD_ROUTING_INTERFACE_CHANGE nework event is recorded when the local interface that should be used to reach the destination specified in **WSAIoctl()** with SIO_ROUTING_INTERFACE_CHANGE changes **AFTER** such IOCTL has been issued.

The FD_ADDRESS_LIST_CHANGE network event is recorded when the list of addresses of socket's protocol family to which the application can bind changes **AFTER WSAIoctl()** with SIO_ADDRESS_LIST_CHANGE has been issued.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.

WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid, or the specified socket is in an invalid state.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function).
WSAENOTSOCK	The descriptor is not a socket.

See Also **WSAAsyncSelect(), WSACloseEvent(), WSACreateEvent(), WSAEnumNetworkEvents(), WSAWaitForMultipleEvents().**

4.33. WSAGetLastError()

Description Get the error status for the last operation which failed.

#include <winsock2.h>

int WSAAPI WSAGetLastError (void);

Remarks This function returns the last network error that occurred. When a particular WinSock function indicates that an error has occurred, this function should be called to retrieve the appropriate error code. This error code may be different from the error code obtained from **getsockopt()** SO_ERROR—which is socket-specific--since **WSAGetLastError()** is for all sockets (i.e., thread-specific)..

A successful function call, or a call to **WSAGetLastError(),** does not reset the error code. To reset the error code, use the **WSASetLastError()** function call with *iError* set to zero (NOTE: **getsockopt()** SO_ERROR also resets the error code to zero)

This function should not be used to check for an error value on receipt of an asynchronous message. In this case the error value is passed in the *lParam* field of the message, and this may differ from the value returned by **WSAGetLastError().**

Return Value The return value indicates the error code for this thread's last WinSock operation that failed.

See Also **WSASetLastError(), getsockopt()**

4.34. WSAGetOverlappedResult()

Description Returns the results of an overlapped operation on the specified socket.

```
#include <winsock2.h>

BOOL WSAAPI
WSAGetOverlappedResult (
        IN      SOCKET          s,
        IN      LPWSAOVERLAPPED lpOverlapped,
        OUT     LPDWORD         lpcbTransfer,
        IN      BOOL            fWait,
        OUT     LPDWORD         lpdwFlags
);
```

s Identifies the socket. This is the same socket that was specified when the overlapped operation was started by a call to **WSARecv(), WSARecvFrom(), WSASend(), WSASendTo()**, or **WSAIoctl()**.

lpOverlapped Points to a WSAOVERLAPPED structure that was specified when the overlapped operation was started.

lpcbTransfer Points to a 32-bit variable that receives the number of bytes that were actually transferred by a send or receive operation, or by **WSAIoctl()**.

fWait Specifies whether the function should wait for the pending overlapped operation to complete. If TRUE, the function does not return until the operation has been completed. If FALSE and the operation is still pending, the function returns FALSE and the **WSAGetLastError()** function returns WSA_IO_INCOMPLETE. The *fWait* parameter may be set to TRUE only if the overlapped operation selected event-based completion notification.

lpdwFlags Points to a 32-bit variable that will receive one or more flags that supplement the completion status. If the overlapped operation was initiated via **WSARecv()** or **WSARecvFrom()**, this parameter will contain the results value for *lpFlags* parameter.

Remarks The results reported by the **WSAGetOverlappedResult()** function are those of the specified socket's last overlapped operation to which the specified WSAOVERLAPPED structure was provided, and for which the operation's results were pending. A pending operation is indicated when the function that started the operation returns SOCKET_ERROR, and the **WSAGetLastError()** function returns WSA_IO_PENDING. When an I/O operation is pending, the function that started the operation resets the *hEvent* member of the WSAOVERLAPPED structure to the nonsignaled state. Then when the pending operation has been completed, the system sets the event object to the signaled state.

If the *fWait* parameter is TRUE, **WSAGetOverlappedResult()** determines whether the pending operation has been completed by waiting for the event object to be in the signaled state. A client may set *fWait* parameter to TRUE only if it selected event-based completion notification when the IO operation was requested. If another form of notification was selected, the usage of the *hEvent* member of the WSAOVERLAPPED structure is different, and setting *fWait* to TRUE causes unpredictable results.

Return Value If **WSAGetOverlappedResult()** succeeds, the return value is TRUE. This means that the overlapped operation has completed successfully and that the value pointed to by *lpcbTransfer* has been updated. If **WSAGetOverlappedResult()** returns FALSE, this means that either the overlapped operation has not completed or the overlapped operation completed but with errors, or that completion status could not be determined due to errors in one or more parameters to **WSAGetOverlappedResult()**. On failure, the value pointed to by *lpcbTransfer* will not be updated. Use **WSAGetLastError()** to determine the cause of the failure (either of **WSAGetOverlappedResult()** or of the associated overlapped operation).

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSA_INVALID_HANDLE	The *hEvent* field of the WSAOVERLAPPED structure does not contain a valid event object handle.
WSA_INVALID_PARAMETER	One of the parameters is unacceptable.
WSA_IO_INCOMPLETE	*fWait* is FALSE and the I/O operation has not yet completed.
WSAEFAULT	One or more of the *lpOverlapped, lpcbTransfer, or lpdwFlags* arguments are not a valid part of the user address space.

See Also **WSACreateEvent(), WSAWaitForMultipleEvents(), WSARecv(), WSARecvFrom(), WSASend(), WSASendTo(), WSAConnect(), WSAAccept(), WSAIoctl().**

4.35. WSAGetQOSByName()

Description Initializes a QOS structure based on a named template, or retrieves an enumeration of the available template names.

```
#include <winsock2.h>

BOOL WSAAPI
WSAGetQOSByName(
        IN          SOCKET          s,
        IN OUT      LPWSABUF        lpQOSName,
        OUT         LPQOS           lpQOS
);
```

s A descriptor identifying a socket.

lpQOSName Specifies the QOS template name, or supplies a buffer to retrieve an enumeration of the available template names.

lpQOS A pointer to the QOS structure to be filled.

Remarks Applications may use this function to initialize a QOS structure to a set of known values appropriate for a particular service class or media type. These values are stored in a template which is referenced by a well-known name. The client may retrieve these values by setting the *buf* member of the WSABUF indicated by *lpQOSName* to point to a string of non-zero length specifying a template name. In this case the usage of *lpQOSName* is IN only, and results are returned through *lpQOS*.

Alternatively, the client may use this function to retrieve an enumeration of available template names. The client may do this by setting the *buf* member of the WSABUF indicated by *lpQOSName* to a zero-length null-terminated string. In this case the buffer indicated by *buf* is over-written with a sequence of as many null-terminated template names are available up to the number of bytes available in *buf* as indicated by the *len* member of the WSABUF indicated by *lpQOSName*. The list of names itself is terminated by a zero-length name. When **WSAGetQOSByName()** is used to retrieve template names, the *lpQOS* parameter is ignored.

Return Value If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE. To get extended error information, call **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lpQOSName or lpQOS* arguments are not a valid part of the user address space, or the buffer length for *lpQOS* is too small.
WSAEINVAL	The specified QOS template name is invalid.

See Also **WSAConnect(), WSAAccept(), getsockopt().**

4.36. WSAHtonl()

Description Convert a **u_long** from host byte order to network byte order.

```
#include <winsock2.h>

int WSAAPI
WSAHtonl (
        IN      SOCKET          s,
        IN      u_long          hostlong,
        OUT     u_long FAR *    lpnetlong
);
```

s A descriptor identifying a socket.

hostlong A 32-bit number in host byte order.

lpnetlong A pointer to a 32-bit number in network byte order.

Remarks This routine takes a 32-bit number in host byte order and returns a 32-bit number pointed to by the *lpnetlong* parameter in the network byte order associated with socket *s*.

Return Value If no error occurs, **WSAHtonl()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lpnetlong* argument is not totally contained in a valid part of the user address space.

See Also **htonl()**, **htons()**, **ntohs(), ntohl(), WSAHtons(), WSANtohl(), WSANtohs()**.

4.37. WSAHtons()

Description Convert a **u_short** from host byte order to network byte order.

```
#include <winsock2.h>

int WSAAPI
WSAHtons (
        IN      SOCKET          s,
        IN      u_short         hostshort,
        OUT     u_short FAR *   lpnetshort
);
```

s A descriptor identifying a socket.

hostshort A 16-bit number in host byte order.

lpnetshort A pointer to a 16-bit number in network byte order.

Remarks This routine takes a 16-bit number in host byte order and returns a 16-bit number pointed to by the *lpnetshort* parameter in the network byte order associated with socket *s*.

Return Value If no error occurs, **WSAHtons()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lpnetshort* argument is not totally contained in a valid part of the user address space.

See Also **htonl()**, **htons()**, **ntohs(), ntohl(), WSAHtonl(), WSANtohl(), WSANtohs()**.

4.38. WSAIoctl()

Description Control the mode of a socket.

```
#include <winsock2.h>

int WSAAPI
WSAIoctl (
        IN      SOCKET                          s,
        IN      DWORD                           dwIoControlCode,
        IN      LPVOID                          lpvInBuffer,
        IN      DWORD                           cbInBuffer,
        OUT     LPVOID                          lpvOutBuffer,
        IN      DWORD                           cbOutBuffer,
        OUT     LPDWORD                         lpcbBytesReturned,
        IN      LPWSAOVERLAPPED                 lpOverlapped,
        IN      LPWSAOVERLAPPED_COMPLETION_ROUTINE
                                                lpCompletionRoutine
);
```

s	Handle to a socket
dwIoControlCode	Control code of operation to perform
lpvInBuffer	Address of input buffer
cbInBuffer	Size of input buffer
lpvOutBuffer	Address of output buffer
cbOutBuffer	Size of output buffer
lpcbBytesReturned	Address of actual bytes of output
lpOverlapped	Address of WSAOVERLAPPED structure (ignored for non-overlapped sockets)
lpCompletionRoutine	A pointer to the completion routine called when the operation has been completed. (ignored for non-overlapped sockets)

Remarks This routine is used to set or retrieve operating parameters associated with the socket, the transport protocol, or the communications subsystem.

If both *lpOverlapped* and *lpCompletionRoutine* are NULL, the socket in this function will be treated as a non-overlapped socket.

For non-overlapped socket, *lpOverlapped* and *lpCompletionRoutine* parameters are ignored, and this function behaves like the standard **ioctlsocket()** function except that it may block if socket *s* is in the blocking mode. Note that if socket *s* is in the non-blocking mode, this function may return WSAEWOULDBLOCK if the specified operation cannot be finished immediately. In this case, the application may change the socket to the blocking mode and reissue the request or wait for the corresponding network event (such as FD_ROUTING_INTERFACE_CHANGE or FD_ADDRESS_LIST_CHANGE in case of SIO_ROUTING_INTERFACE_CHANGE

or SIO_ADDRESS_LIST_CHANGE) using Windows message (via **WSAAsyncSelect()**) or event (via **WSAEventSelect()**) based notification mechanism. For overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The final completion status is retrieved via **WSAGetOverlappedResult()**. The *lpcbBytesReturned* parameter is ignored.

Any ioctl may block indefinitely, depending on the service provider's implementation. If the application cannot tolerate blocking in a **WSAIoctl()** call, overlapped I/O would be advised for ioctls that are especially likely to block including:

SIO_FINDROUTE
SIO_FLUSH
SIO_GET_QOS
SIO_GET_GROUP_QOS
SIO_SET_QOS
SIO_SET_GROUP_QOS
SIO_ROUTING_INTERFACE_CHANGE
SIO_ADDRESS_LIST_CHANGE

Some protocol-specific ioctls may also be especially likely to block. Check the relevant protocol-specific annex for any available information.

In as much as the *dwIoControlCode* parameter is now a 32 bit entity, it is possible to adopt an encoding scheme that preserves the currently defined **ioctlsocket()** opcodes while providing a convenient way to partition the opcode identifier space. The *dwIoControlCode* parameter is architected to allow for protocol and vendor independence when adding new control codes, while retaining backward compatibility with the Windows Sockets 1.1 and Unix control codes. The *dwIoControlCode* parameter has the following form:

3 1	3 0	2 9	2 2 8 7	2 2 2 2 2 2 2 2 1 1 1 1 6 5 4 3 2 1 0 9 8 7 6	1 1 1 1 1 1 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
I	**O**	**V**	**T**	**Vendor/Address Family**	**Code**

I is set if the input buffer is valid for the code, as with **IOC_IN**.

O is set if the output buffer is valid for the code, as with **IOC_OUT**. Note that for codes with both input and output parameters, both **I** and **O** will be set.

V is set if there are no parameters for the code, as with **IOC_VOID**.

T is a two-bit quantity which defines the type of ioctl. The following values are defined:

0 - The ioctl is a standard Unix ioctl code, as with FIONREAD, FIONBIO, etc.

1 - The ioctl is a generic Windows Sockets 2 ioctl code. New ioctl codes defined for Windows Sockets 2 will have **T** == **1**.

2 - The ioctl applies only to a specific address family.

3 - The ioctl applies only to a specific vendor's provider. This type allows companies to be assigned a vendor number which appears in the

Vendor/Address Family field, and then the vendor can define new ioctls specific to that vendor without having to register the ioctl with a clearinghouse, thereby providing vendor flexibility and privacy.

Vendor/Address Family- An 11-bit quantity which defines the vendor who owns the code (if **T** == **3**) or which contains the address family to which the code applies (if **T** == **2**). If this is a Unix ioctl code (**T** == **0**) then this field has the same value as the code on Unix. If this is a generic Windows Sockets 2 ioctl (**T** == **1**) then this field can be used as an extension of the "code" field to provide additional code values.

Code - A 16-bit quantity that contains the specific ioctl code for the operation (i.e. the command).

The following Unix ioctl codes (commands) are supported:

Command	Semantics
FIONBIO	Enable or disable non-blocking mode on socket *s*. *lpvInBuffer* points at an **unsigned long**, which is non-zero if non-blocking mode is to be enabled and zero if it is to be disabled. When a socket is created, it operates in blocking mode (i.e. non-blocking mode is disabled). This is consistent with BSD sockets. The **WSAAsyncSelect()** or **WSAEventSelect()** routine automatically sets a socket to nonblocking mode. If **WSAAsyncSelect()** or **WSAEventSelect()** has been issued on a socket, then any attempt to use **WSAIoctl()** to set the socket back to blocking mode will fail with WSAEINVAL. To set the socket back to blocking mode, an application must first disable **WSAAsyncSelect()** by calling **WSAAsyncSelect()** with the *lEvent* parameter equal to 0, or disable **WSAEventSelect()** by calling **WSAEventSelect()** with the *lNetworkEvents* parameter equal to 0.
FIONREAD	Determine the amount of data which can be read atomically from socket *s*. *lpvOutBuffer* points at an **unsigned long** in which **WSAIoctl()** stores the result. If *s* is stream-oriented (e.g., type SOCK_STREAM), FIONREAD returns the total amount of data which may be read in a single receive operation; this is normally the same as the total amount of data queued on the socket, but since data stream is byte-oriented, this is not guaranteed. If *s* is message-oriented (e.g., type SOCK_DGRAM), FIONREAD returns the size of the first datagram (message) queued on the socket.
SIOCATMARK	Determine whether or not all out-of-band data has been read. This applies only to a socket of stream style (e.g., type SOCK_STREAM) which has been configured for in-line reception of any out-of-band data (SO_OOBINLINE). If no out-of-band data is waiting to be read, the operation returns TRUE. Otherwise it returns FALSE, and the next receive operation performed on the socket will retrieve some or all of the data preceding the "mark"; the application should use the SIOCATMARK operation to determine whether any remains. If there is any normal data preceding the "urgent" (out of band) data, it will be received in order. (Note that receive operations will never mix out-of-

band and normal data in the same call.) *lpvOutBuffer* points at a **BOOL** in which **WSAIoctl()** stores the result.

The following WinSock 2 commands are supported:

Command Semantics

SIO_ASSOCIATE_HANDLE (opcode setting: I, T==1)

Associate this socket with the specified handle of a companion interface. The input buffer contains the integer value corresponding to the manifest constant for the companion interface (e.g., TH_NETDEV, TH_TAPI, etc.), followed by a value which is a handle of the specified companion interface, along with any other required information. Refer to the appropriate section in the ***Windows Sockets 2 Protocol-Specific Annex*** for details specific to a particular companion interface. The total size is reflected in the input buffer length. No output buffer is required. The WSAENOPROTOOPT error code is indicated for service providers which do not support this ioctl. The handle associated by this ioctl can be retrieved using SIO_TRANSLATE_HANDLE.

A companion interface might be used, for example, if a particular provider provides (1) a great deal of additional controls over the behavior of a socket and (2) the controls are provider-specific enough that they don't map to existing WinSock functions or ones likely to be defined in the future. It is recommend that the Component Object Model (COM) be used instead of this ioctl to discover and track other interfaces that might be supported by a socket. This ioctl is present for (reverse) compatibility with systems where COM is not available or cannot be used for some other reason.

SIO_ENABLE_CIRCULAR_QUEUEING (opcode setting: V, T==1)

Indicates to the underlying message-oriented service provider that a newly arrived message should never be dropped because of a buffer queue overflow. Instead, the oldest message in the queue should be eliminated in order to accommodate the newly arrived message. No input and output buffers are required. Note that this ioctl is only valid for sockets associated with unreliable, message-oriented protocols. The WSAENOPROTOOPT error code is indicated for service providers which do not support this ioctl.

SIO_FIND_ROUTE (opcode setting: O, T==1)

When issued, this ioctl requests that the route to the remote address specified as a sockaddr in the input buffer be discovered. If the address already exists in the local cache, its entry is invalidated. In the case of Novell's IPX, this call initiates an IPX GetLocalTarget (GLT), which queries the network for the given remote address.

SIO_FLUSH (opcode setting: V, T==1)

Discards current contents of the sending queue associated with this socket. No input and output buffers are required. The WSAENOPROTOOPT error code is indicated for service providers which do not support this ioctl.

SIO_GET_BROADCAST_ADDRESS (opcode setting: O, T==1)

This ioctl fills the output buffer with a sockaddr struct containing a suitable broadcast address for use with **sendto()**/**WSASendTo()**.

SIO_GET_EXTENSION_FUNCTION_POINTER (opcode setting: O, I, T==1)

Retrieve a pointer to the specified extension function supported by the associated service provider. The input buffer contains a globally unique identifier (GUID) whose value identifies the extension function in question. The pointer to the desired function is returned in the output buffer. Extension function identifiers are established by service provider vendors and should be included in vendor documentation that describes extension function capabilities and semantics.

SIO_GET_QOS (opcode setting: O, I,T==1)

Retrieve the QOS structure associated with the socket. The input buffer is optional. Some protocols (e.g. RSVP) allow the input buffer to be used to qualify a QOS request. The QOS structure will be copied into the output buffer. The output buffer must be sized large enough to be able to contain the full QOS struct. The WSAENOPROTOOPT error code is indicated for service providers which do not support QOS.

SIO_GET_GROUP_QOS (opcode setting: O, I,T==1)

Reserved for future use with socket groups: Retrieve the QOS structure associated with the socket group to which this socket belongs. The input buffer is optional. Some protocols (e.g. RSVP) allow the input buffer to be used to qualify a QOS request. The QOS structure will be copied into the output buffer. If this socket does not belong to an appropriate socket group, the *SendingFlowspec* and *ReceivingFlowspec* fields of the returned QOS struct are set to NULL. The WSAENOPROTOOPT error code is indicated for service providers which do not support QOS.

SIO_MULTIPOINT_LOOPBACK (opcode setting: I, T==1)

Controls whether data sent in a multipoint session will also be received by the same socket on the local host. A value of TRUE causes loopback reception to occur while a value of FALSE prohibits this. By default, loopback is *enabled*.

SIO_MULTICAST_SCOPE (opcode setting: I, T==1)

Specifies the scope over which multicast transmissions will occur. Scope is defined as the number of routed network segments to be covered. A scope of zero would indicate that the multicast transmission would not be placed “on the wire” but could be disseminated across sockets within the local host. A scope value of one (the default) indicates that the transmission will be placed on the wire, but will not cross any routers. Higher scope values determine the number of routers that may be crossed. Note that this corresponds to the time-to-live (TTL) parameter in IP multicasting. By default, scope is 1.

SIO_SET_QOS (opcode setting: I, T==1)

Associate the supplied QOS structure with the socket. No output

buffer is required, the QOS structure will be obtained from the input buffer. The WSAENOPROTOOPT error code is indicated for service providers which do not support QOS.

SIO_SET_GROUP_QOS (opcode setting: I, T==1)

Reserved for future use with socket groups: Establish the supplied QOS structure with the socket group to which this socket belongs. No output buffer is required, the QOS structure will be obtained from the input buffer. The WSAENOPROTOOPT error code is indicated for service providers which do not support QOS, or if the socket descriptor specified is not the creator of the associated socket group.

SIO_TRANSLATE_HANDLE (opcode setting: I, O, T==1)

To obtain a corresponding handle for socket *s* that is valid in the context of a companion interface (e.g., TH_NETDEV, TH_TAPI, etc.). A manifest constant identifying the companion interface along with any other needed parameters are specified in the input buffer. The corresponding handle will be available in the output buffer upon completion of this function. Refer to the appropriate section in the ***Windows Sockets 2 Protocol-Specific Annex*** for details specific to a particular companion interface. The WSAENOPROTOOPT error code is indicated for service providers which do not support this ioctl for the specified companion interface. This ioctl retrieves the handle associated using SIO_TRANSLATE_HANDLE.

It is recommend that the Component Object Model (COM) be used instead of this ioctl to discover and track other interfaces that might be supported by a socket. This ioctl is present for (reverse) compatibility with systems where COM is not available or cannot be used for some other reason.

SIO_ROUTING_INTERFACE_QUERY (opcode setting: I, O, T==1)

To obtain the address of the local interface (represented as SOCKADDR structure) which should be used to send to the remote address specified in the input buffer (as SOCKADDR). Remote multicast addresses may be submitted in the input buffer to get the address of the preferred interface for multicast transmission. In any case, the interface address returned may be used by the application in a subsequent bind() request.

Note that routes are subject to change. Therefore, applications cannot rely on the information returned by SIO_ROUTING_INTERFACE_QUERY to be persistent. Applications may register for routing change notifications via the SIO_ROUTING_INTERFACE_CHANGE IOCTL which provides for notification via either overlapped IO or FD_ROUTING_INTERFACE_CHANGE event. The following sequence of actions can be used to guarantee that the application always has current routing interface information for a given destination:

- issue SIO_ROUTING_INTERFACE_CHANGE IOCTL
- issue SIO_ROUTING_INTERFACE_QUERY IOCTL

- whenever SIO_ROUTING_INTERFACE_CHANGE IOCTL notifies the application of routing change (either via overlapped IO or by signaling FD_ROUTING_INTERFACE_CHANGE event), the whole sequence of actions should be repeated.

If output buffer is not large enough to contain the interface address, SOCKET_ERROR is returned as the result of this IOCTL and **WSAGetLastError()** returns WSAEFAULT. The required size of the output buffer will be returned in *lpcbBytesReturned* in this case. Note the WSAEFAULT error code is also returned if the *lpvInBuffer*, *lpvOutBuffer* or *lpcbBytesReturned* parameter is not totally contained in a valid part of the user address space.

If the destination address specified in the input buffer cannot be reached via any of the available interfaces, SOCKET_ERROR is returned as the result of this IOCTL and **WSAGetLastError()** returns WSAENETUNREACH or even WSAENETDOWN if all of the network connectivity is lost.

SIO_ROUTING_INTERFACE_CHANGE (opcode setting: I, T==1)

To receive notification of the interface change that should be used to reach the remote address in the input buffer (specified as a SOCKADDR structure). No output information will be provided upon completion of this IOCTL; the completion merely indicates that routing interface for a given destination has changed and should be queried again via SIO_ROUTING_INTERFACE_QUERY.

It is assumed (although not required) that the application uses overlapped IO to be notified of routing interface change via completion of SIO_ROUTING_INTERFACE_CHANGE request. Alternatively, if the SIO_ROUTING_INTERFACE_CHANGE IOCTL is issued on non-blocking socket AND without overlapped parameters (lpOverlapped / CompletionRoutine are set NULL), it will complete immediately with error WSAEWOULDBLOCK, and the application can then wait for routing change events via call to **WSAEventSelect()** or **WSAAsyncSelect()** with FD_ROUTING_INTERFACE_CHANGE bit set in the network event bitmask

It is recognized that routing information remains stable in most cases so that requiring the application to keep multiple outstanding IOCTLs to get notifications about all destinations that it is interested in as well as having service provider to keep track of all them will unnecessarily tie significant system resources. This situation can be avoided by extending the meaning of the input parameters and relaxing the service provider requirements as follows:

- the application can specify a protocol family specific wildcard address (same as one used in **bind()** call when requesting to bind to any available address) to request notifications of any routing changes. This allows the application to keep only one outstanding SIO_ROUTING_INTERFACE_CHANGE for all the sockets/destinations it has and then use

SIO_ROUTING_INTERFACE_QUERY to get the actual routing information

- service provider can opt to ignore the information supplied by the application in the input buffer of the SIO_ROUTING_INTERFACE_CHANGE (as though the application specified a wildcard address) and complete the SIO_ROUTING_INTERFACE_CHANGE IOCTL or signal FD_ROUTING_INTERFACE_CHANGE event in the event of any routing information change (not just the route to the destination specified in the input buffer).

SIO_ADDRESS_LIST_QUERY (opcode setting: I, O, T==1)

To obtain a list of local transport addresses of socket's protocol family to which the application can bind. The list returned in the output buffer using the following format:

```
typedef struct _SOCKET_ADDRESS_LIST {
      INT               iAddressCount;
      SOCKET_ADDRESS    Address[1];
} SOCKET_ADDRESS_LIST, FAR *
LPSOCKET_ADDRESS_LIST;
```

Members:

iAddressCount - number of address structures in the list;
Address - array of protocol family specific address structures.

Note that in Win32 Plug-n-Play environments addresses can be added/removed dynamically. Therefore, applications cannot rely on the information returned by SIO_ADDRESS_LIST_QUERY to be persistent. Applications may register for address change notifications via the SIO_ADDRESS_LIST_CHANGE IOCTL which provides for notification via either overlapped IO or FD_ADDRESS_LIST_CHANGE event. The following sequence of actions can be used to guarantee that the application always has current address list information:

- issue SIO_ADDRESS_LIST_CHANGE IOCTL
- issue SIO_ADDRESS_LIST_QUERY IOCTL
- whenever SIO_ADDRESS_LIST_CHANGE IOCTL notifies the application of address list change (either via overlapped IO or by signaling FD_ADDRESS_LIST_CHANGE event), the whole sequence of actions should be repeated.

If output buffer is not large enough to contain the address list, SOCKET_ERROR is returned as the result of this IOCTL and **WSAGetLastError()** returns WSAEFAULT. The required size of the output buffer will be returned in *lpcbBytesReturned* in this case. Note the WSAEFAULT error code is also returned if the *lpvInBuffer*, *lpvOutBuffer* or *lpcbBytesReturned* parameter is not totally contained in a valid part of the user address space.

SIO_ADDRESS_LIST_CHANGE (opcode setting: T==1)

To receive notification of changes in the list of local transport addresses of socket's protocol family to which the application can bind. No output information will be provided upon completion of this

IOCTL; the completion merely indicates that list of available local address has changed and should be queried again via SIO_ADDRESS_LIST_QUERY.

It is assumed (although not required) that the application uses overlapped IO to be notified of change via completion of SIO_ADDRESS_LIST_CHANGE request. Alternatively, if the SIO_ADDRESS_LIST_CHANGE IOCTL is issued on non-blocking socket AND without overlapped parameters (*lpOverlapped / lpCompletionRoutine* are set to NULL), it will complete immediately with error WSAEWOULDBLOCK. The application can then wait for address list change events via call to **WSAEventSelect()** or **WSAAsyncSelect()** with FD_ADDRESS_LIST_CHANGE bit set in the network event bitmask.

If an overlapped operation completes immediately, this function returns a value of zero and the *lpcbBytesReturned* parameter is updated with the number of bytes in the output buffer. If the overlapped operation is successfully initiated and will complete later, this function returns SOCKET_ERROR and indicates error code WSA_IO_PENDING. In this case, *lpcbBytesReturned* is not updated. When the overlapped operation completes the amount of data in the output buffer is indicated either via the *cbTransferred* parameter in the completion routine (if specified), or via the *lpcbTransfer* parameter in **WSAGetOverlappedResult()**.

When called with an overlapped socket, the *lpOverlapped* parameter must be valid for the duration of the overlapped operation. The WSAOVERLAPPED structure has the following form:

```
typedef struct _WSAOVERLAPPED {
      DWORD       Internal;         // reserved
      DWORD       InternalHigh;     // reserved
      DWORD       Offset;           // reserved
      DWORD       OffsetHigh;       // reserved
      WSAEVENT    hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* field of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use **WSAWaitForMultipleEvents()** or **WSAGetOverlappedResult()** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* field is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-NULL *lpCompletionRoutine* and later calls **WSAGetOverlappedResult()** for the same overlapped IO request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult()** to TRUE. In this case the usage of the *hEvent* field is undefined, and attempting to wait on the *hEvent* field would produce unpredictable results.

The prototype of the completion routine is as follows:

void CALLBACK

```
CompletionRoutine(
        IN      DWORD               dwError,
        IN      DWORD               cbTransferred,
        IN      LPWSAOVERLAPPED     lpOverlapped,
        IN      DWORD               dwFlags
);
```

CompletionRoutine is a placeholder for an application-defined or library-defined function. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes returned. Currently there are no flag values defined and *dwFlags* will be zero. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. The completion routines may be called in any order, not necessarily in the same order the overlapped operations are completed.

Compatibility

The ioctl codes with **T** == **0** are a subset of the ioctl codes used in Berkeley sockets. In particular, there is no command which is equivalent to FIOASYNC.

Return Value

Upon successful completion, the **WSAIoctl ()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *lpvInBuffer, lpvOutBuffer lpcbBytesReturned, lpOverlapped, or lpCompletionRoutine* argument is not totally contained in a valid part of the user address space, or the *cbInBuffer* or *cbOutBuffer* argument is too small.
WSAEINVAL	*dwIoControlCode* is not a valid command, or a supplied input parameter is not acceptable, or the command is not applicable to the type of socket supplied.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAENOTSOCK	The descriptor *s* is not a socket.
WSAEOPNOTSUPP	The specified ioctl command cannot be realized, e.g., the flow specs specified in SIO_SET_QOS or SIO_SET_GROUP_QOS cannot be satisfied.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSAEWOULDBLOCK	The socket is marked as non-blocking and the requested operation would block.

See Also **socket()**, **ioctlsocket()**, **WSASocket()**, **setsockopt()**, **getsockopt().**

4.39. WSAIsBlocking()

Description Determine if a blocking call is in progress. **WSAIsBlocking()** is only available for WinSock 1.1 apps (that is, if at least one thread within the process negotiates version 1.0 or 1.1 at **WSAStartup()**).

Important Note: *This function is for backwards compatibility with WinSock 1.1 and is not considered part of the WinSock 2 specification. WinSock 2 applications should* ***not*** *use this function.*

#include <winsock2.h>

BOOL WSAAPI WSAIsBlocking (void);

Remarks This function allows a WinSock 1.1 application to determine if it is executing while waiting for a previous blocking call to complete.

Return Value The return value is TRUE if there is an outstanding blocking function awaiting completion in the current thread. Otherwise, it is FALSE.

Comments In 16-bit WinSock 1.1 environments, although a call issued on a blocking socket appears to an application program as though it "blocks", the WinSock DLL has to relinquish the processor to allow other applications to run. This means that it is possible for the application which issued the blocking call to be re-entered, depending on the message(s) it receives. In this instance, the **WSAIsBlocking()** function can be used to ascertain whether the task has been re-entered while waiting for an outstanding blocking call to complete. Note that WinSock 1.1 prohibits more than one outstanding call per thread.

See Also **WSACancelBlockingCall(), WSASetBlockingHook(), WSAUnhookBlockingHook()**

4.40. WSAJoinLeaf()

Description Join a leaf node into a multipoint session, exchange connect data, and specify needed quality of service based on the supplied flow specs.

```
#include <winsock2.h>

SOCKET WSAAPI
WSAJoinLeaf (
        IN      SOCKET                      s,
        IN      const struct sockaddr FAR * name,
        IN      int                         namelen,
        IN      LPWSABUF                    lpCallerData,
        OUT     LPWSABUF                    lpCalleeData,
        IN      LPQOS                       lpSQOS,
        IN      LPQOS                       lpGQOS,
        IN      DWORD                       dwFlags
);
```

s	A descriptor identifying a multipoint socket.
name	The name of the peer to which the socket is to be joined.
namelen	The length of the *name*.
lpCallerData	A pointer to the user data that is to be transferred to the peer during multipoint session establishment.
lpCalleeData	A pointer to the user data that is to be transferred back from the peer during multipoint session establishment.
lpSQOS	A pointer to the flow specs for socket *s*, one for each direction.
lpGQOS	Reserved for future use with socket groups: A pointer to the flow specs for the socket group (if applicable).
dwFlags	Flags to indicate that the socket is acting as a sender, receiver, or both.

Remarks This function is used to join a leaf node to a multipoint session, and to perform a number of other ancillary operations that occur at session join time as well. If the socket, *s*, is unbound, unique values are assigned to the local association by the system, and the socket is marked as bound.

WSAJoinLeaf() has the same parameters and semantics as **WSAConnect()** except that it returns a socket descriptor (as in **WSAAccept())**, and it has an additional *dwFlags* parameter. Only multipoint sockets created using **WSASocket()** with appropriate multipoint flags set may be used for input parameter *s* in this function. The returned socket descriptor will not be useable until after the join operation completes (e.g. if the socket is in non-blocking mode, after a corresponding FD_CONNECT indication has been received from **WSAAsyncSelect()** or **WSAEventSelect()** on the original socket *s*), except that **closesocket()** may be invoked on this new socket descriptor to cancel a pending join operation. A root application in a multipoint session may call

WSAJoinLeaf() one or more times in order to add a number of leaf nodes, however at most one multipoint connection request may be outstanding at a time. Refer to Appendix B. Multipoint and Multicast Semantics for additional information.

For non-blocking sockets it is often not possible to complete the connection immediately. In such a case, this function returns an as-yet unusable socket descriptor and the operation proceeds. There is no error code such as WSAEWOULDBLOCK in this case, since the function has effectively returned a "successful start" indication. When the final outcome success or failure becomes known, it may be reported through **WSAAsyncSelect()** or **WSAEventSelect()** depending on how the client registers for notification on the original socket *s*. In either case, the notification is announced with FD_CONNECT and the error code associated with the FD_CONNECT indicates either success or a specific reason for failure. Note that **select()** cannot be used to detect completion notification for **WSAJoinLeaf()**.

The socket descriptor returned by **WSAJoinLeaf()** is different depending on whether the input socket descriptor, *s*, is a c_root or a c_leaf. When used with a c_root socket, the *name* parameter designates a particular leaf node to be added and the returned socket descriptor is a c_leaf socket corresponding to the newly added leaf node. The newly created socket has the same properties as *s* including asynchronous events registered with **WSAAsyncSelect()** or with **WSAEventSelect()**, but **not** including the c_root socket's group ID, if any. It is not intended to be used for exchange of multipoint data, but rather is used to receive network event indications (e.g. FD_CLOSE) for the connection that exists to the particular c_leaf. Some multipoint implementations may also allow this socket to be used for "side chats" between the root and an individual leaf node. An FD_CLOSE indication will be received for this socket if the corresponding leaf node calls **closesocket()** to drop out of the multipoint session. Symmetrically, invoking **closesocket()** on the c_leaf socket returned from **WSAJoinLeaf()** will cause the socket in the corresponding leaf node to get FD_CLOSE notification.

When **WSAJoinLeaf()** is invoked with a c_leaf socket, the *name* parameter contains the address of the root application (for a rooted control scheme) or an existing multipoint session (non-rooted control scheme), and the returned socket descriptor is the same as the input socket descriptor. In other words, a new socket descriptor is **not** allocated. In a rooted control scheme, the root application would put its c_root socket in the listening mode by calling **listen()**. The standard FD_ACCEPT notification will be delivered when the leaf node requests to join itself to the multipoint session. The root application uses the usual **accept()/WSAAccept()** functions to admit the new leaf node. The value returned from either **accept()** or **WSAAccept()** is also a c_leaf socket descriptor just like those returned from **WSAJoinLeaf()**. To accommodate multipoint schemes that allow both root-initiated and leaf-initiated joins, it is acceptable for a c_root socket that is already in listening mode to be used as an input to **WSAJoinLeaf()**.

The application is responsible for allocating any memory space pointed to directly or indirectly by any of the parameters it specifies.

The *lpCallerData* is a value parameter which contains any user data that is to be sent along with the multipoint session join request. If *lpCallerData* is NULL, no user data will be passed to the peer. The *lpCalleeData* is a result parameter which will contain any user data passed back from the peer as part of the multipoint session establishment. *lpCalleeData->len* initially contains the length of the buffer allocated by the

application and pointed to by *lpCalleeData->buf*. *lpCalleeData->len* will be set to 0 if no user data has been passed back. The *lpCalleeData* information will be valid when the multipoint join operation is complete. For blocking sockets, this will be when the **WSAJoinLeaf()** function returns. For non-blocking sockets, this will be after the join operation has completed (e.g. after FD_CONNECT notification has occurred on the original socket *s*). If *lpCalleeData* is NULL, no user data will be passed back. The exact format of the user data is specific to the address family to which the socket belongs.

At multipoint session establishment time, an application may use the *lpSQOS* and/or *lpGQOS* parameters to override any previous QOS specification made for the socket via **WSAIoctl()** with either the SIO_SET_QOS or SIO_SET_GROUP_QOS opcodes.

lpSQOS specifies the flow specs for socket *s*, one for each direction, followed by any additional provider-specific parameters. If either the associated transport provider in general or the specific type of socket in particular cannot honor the QOS request, an error will be returned as indicated below. The sending or receiving flow spec values will be ignored, respectively, for any unidirectional sockets. If no provider-specific parameters are supplied, the *buf* and *len* fields of *lpSQOS->ProviderSpecific* should be set to NULL and 0, respectively. A NULL value for *lpSQOS* indicates no application supplied QOS.

Reserved for future use with socket groups: *lpGQOS* specifies the flow specs for the socket group (if applicable), one for each direction, followed by any additional provider-specific parameters. If no provider-specific parameters are supplied, the *buf* and *len* fields of *lpGQOS->ProviderSpecific* should be set to NULL and 0, respectively. A NULL value for *lpGQOS* indicates no application-supplied group QOS. This parameter will be ignored if *s* is not the creator of the socket group.

The *dwFlags* parameter is used to indicate whether the socket will be acting only as a sender (JL_SENDER_ONLY), only as a receiver (JL_RECEIVER_ONLY), or both (JL_BOTH).

Comments

When connected sockets break (i.e. become closed for whatever reason), they should be discarded and recreated. It is safest to assume that when things go awry for any reason on a connected socket, the application must discard and recreate the needed sockets in order to return to a stable point.

Return Value

If no error occurs, **WSAJoinLeaf()** returns a value of type SOCKET which is a descriptor for the newly created multipoint socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

On a blocking socket, the return value indicates success or failure of the join operation.

With a non-blocking socket, successful initiation of a join operation is indicated by a return of a valid socket descriptor. Subsequently, an FD_CONNECT indication will be given on the original socket *s* when the join operation completes, either successfully or otherwise. The application must use either **WSAAsyncSelect()** or **WSAEventSelect()** with interest registered for the FD_CONNECT event in order to determine when the join operation has completed and check the associated error code to determine the success or failure of the operation. Note that the **select()** function cannot be used to determine when the join operation completes.

Also, until the multipoint session join attempt completes all subsequent calls to **WSAJoinLeaf()** on the same socket will fail with the error code WSAEALREADY. After the **WSAJoinLeaf()** completes successfully a subsequent attempt will usually fail with the error code WSAEISCONN. An exception to the WSAEISCONN rule occurs for a c_root socket that allows root-initiated joins. In such a case another join may be initiated after a prior **WSAJoinLeaf()** completes.

If the return error code indicates the multipoint session join attempt failed (i.e. WSAECONNREFUSED, WSAENETUNREACH, WSAETIMEDOUT) the application may call **WSAJoinLeaf()** again for the same socket.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The socket's local address is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs at the time of **bind()**, but could be delayed until this function if the **bind()** was to a partially wild-card address (involving ADDR_ANY) and if a specific address needs to be "committed" at the time of this function.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	A non-blocking **WSAJoinLeaf()** call is in progress on the specified socket.
WSAEADDRNOTAVAIL	The remote address is not a valid address (e.g., ADDR_ANY).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to join was forcefully rejected.
WSAEFAULT	The *name* or the *namelen* argument is not a valid part of the user address space, the *namelen* argument is too small, the buffer length for *lpCalleeData*, *lpSQOS*, and *lpGQOS* are too small, or the buffer length for *lpCallerData* is too large.
WSAEISCONN	The socket is already member of the multipoint session.

WSAENETUNREACH	The network can't be reached from this host at this time.
WSAENOBUFS	No buffer space is available. The socket cannot be joined.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	The flow specs specified in *lpSQOS* and *lpGQOS* cannot be satisfied.
WSAEPROTONOSUPPORT	The *lpCallerData* argument is not supported by the service provider.
WSAETIMEDOUT	Attempt to join timed out without establishing a multipoint session.

See Also **accept(), bind(), select(), WSAAccept(), WSAAsyncSelect(), WSAEventSelect(), WSASocket().**

4.41. WSANtohl()

Description Convert a **u_long** from network byte order to host byte order.

```
#include <winsock2.h>

int WSAAPI
WSANtohl (
        IN          SOCKET          s,
        IN          u_long          netlong,
        OUT         u_long FAR *    lphostlong
);
```

s	A descriptor identifying a socket.
netlong	A 32-bit number in network byte order.
lphostlong	A pointer to a 32-bit number in host byte order.

Remarks This routine takes a 32-bit number in the network byte order associated with socket *s* and returns a 32-bit number pointed to by the *lphostlong* parameter in host byte order.

Return Value If no error occurs, **WSANtohl()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lphostlong* argument is not totally contained in a valid part of the user address space.

See Also **ntohl(), htonl()**, **htons()**, **ntohs(), WSAHtonl(), WSAHtons(), WSANtohs()**.

4.42. WSANtohs()

Description Convert a **u_short** from network byte order to host byte order.

```
#include <winsock2.h>

int WSAAPI
WSANtohs (
        IN          SOCKET          s,
        IN          u_short         netshort,
        OUT         u_short FAR *   lphostshort
);
```

s A descriptor identifying a socket.

netshort A 16-bit number in network byte order.

lphostshort A pointer to a 16-bit number in host byte order.

Remarks This routine takes a 16-bit number in the network byte order associated with socket *s* and returns a 16-bit number pointed to by the *lphostshort* parameter in host byte order.

Return Value If no error occurs, **WSANtohs()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lphostshort* argument is not totally contained in a valid part of the user address space.

See Also **htonl()**, **htons()**, **ntohs(), ntohl(), WSAHtonl(), WSAHtons(), WSANtohl()**.

4.43. WSARecv()

Description Receive data from a connected socket

```
#include <winsock2.h>

int WSAAPI
WSARecv (
      IN        SOCKET                              s,
      IN OUT    LPWSABUF                            lpBuffers,
      IN        DWORD                               dwBufferCount,
      OUT       LPDWORD                             lpNumberOfBytesRecvd,
      IN OUT    LPDWORD                             lpFlags,
      IN        LPWSAOVERLAPPED                     lpOverlapped,
      IN        LPWSAOVERLAPPED_COMPLETION_ROUTINE
                                                    lpCompletionRoutine
);
```

s A descriptor identifying a connected socket.

lpBuffers A pointer to an array of **WSABUF** structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer.

```
typedef struct __WSABUF {
      u_long      len;  // buffer length
      char FAR *  buf;  // pointer to buffer
} WSABUF, FAR * LPWSABUF;
```

dwBufferCount The number of **WSABUF** structures in the *lpBuffers* array.

lpNumberOfBytesRecvd A pointer to the number of bytes received by this call if the receive operation completes immediately.

lpFlags A pointer to flags.

lpOverlapped A pointer to a WSAOVERLAPPED structure (ignored for non-overlapped sockets).

lpCompletionRoutine A pointer to the completion routine called when the receive operation has been completed (ignored for non-overlapped sockets).

Remarks This function provides functionality over and above the standard **recv()** function in three important areas:

- It can be used in conjunction with overlapped sockets to perform overlapped receive operations.
- It allows multiple receive buffers to be specified making it applicable to the scatter/gather type of I/O.
- The *lpFlags* parameter is both an INPUT and an OUTPUT parameter, allowing applications to sense the output state of the MSG_PARTIAL flag bit. Note however, that the MSG_PARTIAL flag bit is not supported by all protocols.

WSARecv() is used on connected sockets or bound connectionless sockets specified by the *s* parameter and is used to read incoming data. The socket's local address must be known. For server applications, this is usually done explicitly through **bind()** or implicitly through **accept()** or **WSAAccept()**. Explicit binding is discouraged for client applications. For client applications the socket can become bound implicitly to a local address through **connect(), WSAConnect(), sendto(), WSASendTo(),** or **WSAJoinLeaf()**.

For connected, connectionless sockets, this function restricts the addresses from which received messages are accepted. The function only returns messages from the remote address specified in the connection. Messages from other addresses are (silently) discarded.

For overlapped sockets **WSARecv()** is used to post one or more buffers into which incoming data will be placed as it becomes available, after which the application-specified completion indication (invocation of the completion routine or setting of an event object) occurs. If the operation does not complete immediately, the final completion status is retrieved via the completion routine or **WSAGetOverlappedResult().**

If both *lpOverlapped* and *lpCompletionRoutine* are NULL, the socket in this function will be treated as a non-overlapped socket.

For non-overlapped sockets, the blocking semantics are identical to that of the standard **recv()** function and the *lpOverlapped* and *lpCompletionRoutine* parameters are ignored. Any data which has already been received and buffered by the transport will be copied into the supplied user buffers. For the case of a blocking socket with no data currently having been received and buffered by the transport, the call will block until data is received. WinSock 2 does not define any standard blocking timeout mechanism for this function. For protocols acting as byte-stream protocols the stack tries to return as much data as possible subject to the supplied buffer space and amount of received data available. However, receipt of a single byte is sufficient to unblock the caller. There is no guarantee that more than a single byte will be returned. For protocols acting as message-oriented, a full message is required to unblock the caller.

Whether or not a protocol is acting as byte-stream is determined by the setting of XP1_MESSAGE_ORIENTED and XP1_PSEUDO_STREAM in its WSAPROTOCOL_INFO structure and the setting of the MSG_PARTIAL flag passed in to this function (for protocols that support it). The relevant combinations are summarized in the following table (an asterisk (*) indicates that the setting of this bit does not matter in this case).

XP1_MESSAGE_ ORIENTED	XP1_PSEUDO_ STREAM	MSG_PARTIAL	Acts as
not set	*	*	byte-stream
*	set	*	byte-stream
set	not set	set	byte-stream
set	not set	not set	message-oriented

The supplied buffers are filled in the order in which they appear in the array pointed to by *lpBuffers*, and the buffers are packed so that no holes are created.

The array of **WSABUF** structures pointed to by the *lpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture these **WSABUF** structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays.

For byte stream style sockets (e.g., type SOCK_STREAM), incoming data is placed into the buffers until the buffers are filled, the connection is closed, or internally buffered data is exhausted. Regardless of whether or not the incoming data fills all the buffers, the completion indication occurs for overlapped sockets.

For message-oriented sockets (e.g., type SOCK_DGRAM), an incoming message is placed into the supplied buffers, up to the total size of the buffers supplied, and the completion indication occurs for overlapped sockets. If the message is larger than the buffers supplied, the buffers are filled with the first part of the message. If the MSG_PARTIAL feature is supported by the underlying service provider, the MSG_PARTIAL flag is set in *lpFlags* and subsequent receive operation(s) will retrieve the rest of the message. If MSG_PARTIAL is not supported but the protocol is reliable, **WSARecv()** generates the error WSAEMSGSIZE and a subsequent receive operation with a larger buffer can be used to retrieve the entire message. Otherwise (i.e. the protocol is unreliable and does not support MSG_PARTIAL), the excess data is lost, and **WSARecv()** generates the error WSAEMSGSIZE.

For connection-oriented sockets, **WSARecv()** can indicate the graceful termination of the virtual circuit in one of two ways, depending on whether the socket is a byte stream or message-oriented. For byte streams, zero bytes having been read (as indicated by zero return value to indicate success, and lpNumberOfBytesRecvd value of zero) indicates graceful closure and that no more bytes will ever be read. For message-oriented sockets, where a zero byte message is often allowable, a failure with an error code of WSAEDISCON is used to indicate graceful closure. In any case, a failure with an error code of WSAECONNRESET indicates an abortive close has occurred.

lpFlags may be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *lpFlags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
MSG_PEEK	Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue. This flag is valid only for non-overlapped sockets.
MSG_OOB	Process out-of-band data (See section *3.5. Out-Of-Band data* for a discussion of this topic.)
MSG_PARTIAL	This flag is for message-oriented sockets only. On output, indicates that the data supplied is a portion of the message transmitted by the sender. Remaining portions of the message will be supplied in subsequent receive operations. A subsequent receive operation with MSG_PARTIAL flag cleared indicates end of sender's message. As an input parameter, this flag indicates that the receive operation should complete even if only part of a message has been received by the service provider.

For message-oriented sockets, the MSG_PARTIAL bit is set in the *lpFlags* parameter if a partial message is received. If a complete message is received, MSG_PARTIAL is cleared in *lpFlags*. In the case of delayed completion, the value pointed to by *lpFlags* is not updated. When completion has been indicated the application should call **WSAGetOverlappedResult()** and examine the flags pointed to by the *lpdwFlags* parameter.

Overlapped socket I/O:

If an overlapped operation completes immediately, **WSARecv()** returns a value of zero and the *lpNumberOfBytesRecvd* parameter is updated with the number of bytes received and the flag bits pointed by the *lpFlags* parameter are also updated. If the overlapped operation is successfully initiated and will complete later, **WSARecv()** returns SOCKET_ERROR and indicates error code WSA_IO_PENDING. In this case, *lpNumberOfBytesRecvd* and *lpFlags* are not updated. When the overlapped operation completes the amount of data transferred is indicated either via the *cbTransferred* parameter in the completion routine (if specified), or via the *lpcbTransfer* parameter in **WSAGetOverlappedResult()**. Flag values are obtained by examining the *lpdwFlags* parameter of **WSAGetOverlappedResult().**

This function may be called from within the completion routine of a previous **WSARecv()**, **WSARecvFrom()**, **WSASend()** or **WSASendTo()** function. For a given socket, I/O completion routines will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate overlapped structure. The WSAOVERLAPPED structure has the following form:

```
typedef struct _WSAOVERLAPPED {
      DWORD       Internal;         // reserved
      DWORD       InternalHigh;     // reserved
      DWORD       Offset;           // reserved
      DWORD       OffsetHigh;       // reserved
      WSAEVENT    hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* field of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use **WSAWaitForMultipleEvents()** or **WSAGetOverlappedResult()** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* field is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-NULL *lpCompletionRoutine* and later calls **WSAGetOverlappedResult()** for the same overlapped IO request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult()** to TRUE. In this case the usage of the *hEvent* field is undefined, and attempting to wait on the *hEvent* field would produce unpredictable results.

The completion routine follows the same rules as stipulated for Win32 file I/O completion routines. The completion routine will not be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents()** with the *fAlertable* parameter set to TRUE is invoked.

Transport providers allow an application to invoke send and receive operations from within the context of the socket I/O completion routine, and guarantee that, for a given socket, I/O completion routines will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The prototype of the completion routine is as follows:

```
void CALLBACK
CompletionRoutine(
IN          DWORD               dwError,
IN          DWORD               cbTransferred,
IN          LPWSAOVERLAPPED     lpOverlapped,
IN          DWORD               dwFlags
);
```

CompletionRoutine is a placeholder for an application-defined or library-defined function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes received. *dwFlags* contains information that would have appeared in *lpFlags* if the receive operation had completed immediately. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. When using **WSAWaitForMultipleEvents()**, all waiting completion routines are called before the alertable thread's wait is satisfied with a return code of WSA_IO_COMPLETION. The completion routines may be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order they are supplied.

Return Value

If no error occurs and the receive operation has completed immediately, **WSARecv()** returns 0. Note that in this case the completion routine will have already been scheduled, and to be called once the calling thread is in the alertable state. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAENOTCONN	The socket is not connected.

WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENETRESET	The connection has been broken due to "keep-alive" activity detecting a failure while the operation was in progress.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lpBuffers* argument is not totally contained in a valid part of the user address space.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **WSARecv()** on a socket after **shutdown()** has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	Overlapped sockets: There are too many outstanding overlapped I/O requests. Non-overlapped sockets: The socket is marked as non-blocking and the receive operation cannot be completed immediately.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and (for unreliable protocols only) any trailing portion of the message that did not fit into the buffer has been discarded.
WSAEINVAL	The socket has not been bound (e.g., with **bind()**),**.**
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure.
WSAECONNRESET	The virtual circuit was reset by the remote side.
WSAEDISCON	Socket *s* is message oriented and the virtual circuit was gracefully closed by the remote side.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	The overlapped operation has been canceled due to the closure of the socket.

See Also **WSACloseEvent(),WSACreateEvent(), WSAGetOverlappedResult(), WSASocket(), WSAWaitForMultipleEvents()**

4.44. WSARecvDisconnect()

Description Terminate reception on a socket, and retrieve the disconnect data if the socket is connection-oriented.

```
#include <winsock2.h>

int WSAAPI
WSARecvDisconnect (
        IN      SOCKET      s,
        OUT     LPWSABUF    lpInboundDisconnectData
);
```

s	A descriptor identifying a socket.
lpInboundDisconnectData	A pointer to the incoming disconnect data.

Remarks **WSARecvDisconnect()** is used on connection-oriented sockets to disable reception, and retrieve any incoming disconnect data from the remote party. This is equivalent to a shutdown(SD_RECV), except that **WSASendDisconnect()** also allows receipt of disconnect data (in protocols that support it).

After this function has been successfully issued, subsequent receives on the socket will be disallowed. This has no effect on the lower protocol layers. For TCP sockets, if there is still data queued on the socket waiting to be received, or data arrives subsequently, the connection is reset, since the data cannot be delivered to the user. For UDP, incoming datagrams are accepted and queued. In no case will an ICMP error packet be generated.

To successfully receive incoming disconnect data, an application must use other mechanisms to determine that the circuit has been closed. For example, an application needs to receive an FD_CLOSE notification, or get a 0 return value, or a WSAEDISCON or WSAECONNRESET error code from **recv()**/**WSARecv()**.

Note that **WSARecvDisconnect()** does not close the socket, and resources attached to the socket will not be freed until **closesocket()** is invoked.

Comments **WSARecvDisconnect()** does not block regardless of the SO_LINGER setting on the socket.

An application should not rely on being able to re-use a socket after it has been **WSARecvDisconnect()**ed. In particular, a WinSock provider is not required to support the use of **connect()/WSAConnect()** on such a socket.

Return Value If no error occurs, **WSARecvDisconnect()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.

WSAEFAULT	The buffer referenced by the parameter *lpInboundDisconnectData* is too small.
WSAENOPROTOOPT	The disconnect data is not supported by the indicated protocol family.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.

See Also **connect()**, **socket()**.

4.45. WSARecvFrom()

Description Receive a datagram and store the source address.

```
#include <winsock2.h>

int WSAAPI
WSARecvFrom (
        IN          SOCKET                          s,
        IN OUT      LPWSABUF                        lpBuffers,
        IN          DWORD                           dwBufferCount,
        OUT         LPDWORD                         lpNumberOfBytesRecvd,
        IN OUT      LPDWORD                         lpFlags,
        OUT         struct sockaddr FAR *           lpFrom,
        IN OUT      LPINT                           lpFromlen,
        IN          LPWSAOVERLAPPED                 lpOverlapped,
        IN          LPWSAOVERLAPPED_COMPLETION_ROUTINE
                                                    lpCompletionRoutine
);
```

s A descriptor identifying a socket

lpBuffers A pointer to an array of **WSABUF** structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer.

```
typedef struct __WSABUF {
      u_long      len;  // buffer length
      char FAR *  buf;  // pointer to buffer
} WSABUF, FAR * LPWSABUF;
```

dwBufferCount The number of **WSABUF** structures in the *lpBuffers* array.

lpNumberOfBytesRecvd A pointer to the number of bytes received by this call if the receive operation completes immediately.

lpFlags A pointer to flags.

lpFrom An optional pointer to a buffer which will hold the source address upon the completion of the overlapped operation.

lpFromlen A pointer to the size of the *from* buffer, required only if *lpFrom* is specified.

lpOverlapped A pointer to a WSAOVERLAPPED structure (ignored for non-overlapped sockets).

lpCompletionRoutine A pointer to the completion routine called when the receive operation has been completed (ignored for non-overlapped sockets).

Remarks This function provides functionality over and above the standard **recvfrom()** function in three important areas:

- It can be used in conjunction with overlapped sockets to perform overlapped receive operations.
- It allows multiple receive buffers to be specified making it applicable to the scatter/gather type of I/O.
- The *lpFlags* parameter is both an INPUT and an OUTPUT parameter, allowing applications to sense the output state of the MSG_PARTIAL flag bit. Note however, that the MSG_PARTIAL flag bit is not supported by all protocols.

WSARecvFrom() is used primarily on a connectionless socket specified by *s.* The socket must not be connected. The socket's local address must be known. For server applications, this is usually done explicitly through **bind()**. Explicit binding is discouraged for client applications. For client applications using this function the socket can become bound implicitly to a local address through **sendto(), WSASendTo(),** or **WSAJoinLeaf()**.

For overlapped sockets, this function is used to post one or more buffers into which incoming data will be placed as it becomes available on a (possibly connected) socket, after which the application-specified completion indication (invocation of the completion routine or setting of an event object) occurs. If the operation does not complete immediately, the final completion status is retrieved via the completion routine or **WSAGetOverlappedResult().** Also note that the values pointed to by *lpFrom* and *lpFromlen* are not updated until completion is indicated. Applications must not use or disturb these values until they have been updated, therefore the application must not use automatic (i.e. stack-based) variables for these parameters.

If both *lpOverlapped* and *lpCompletionRoutine* are NULL, the socket in this function will be treated as a non-overlapped socket.

For non-overlapped sockets, the blocking semantics are identical to that of the standard **WSARecv()** function and the *lpOverlapped* and *lpCompletionRoutine* parameters are ignored. Any data which has already been received and buffered by the transport will be copied into the supplied user buffers. For the case of a blocking socket with no data currently having been received and buffered by the transport, the call will block until data is received.

The supplied buffers are filled in the order in which they appear in the array pointed to by *lpBuffers,* and the buffers are packed so that no holes are created.

The array of **WSABUF** structures pointed to by the *lpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture these **WSABUF** structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays.

For connectionless socket types, the address from which the data originated is copied to the buffer pointed by *lpFrom*. The value pointed to by *lpFromlen* is initialized to the size of this buffer, and is modified on completion to indicate the actual size of the address stored there. As noted previously for overlapped sockets, the *lpFrom* and *lpFromlen* parameters are not updated until after the overlapped I/O has completed. The memory pointed to by these parameters must, therefore, remain available to the service provider and cannot be allocated on the application's stack frame. The *lpFrom* and *lpFromlen* parameters are ignored for connection-oriented sockets.

For byte stream style sockets (e.g., type SOCK_STREAM), incoming data is placed into the buffers until the buffers are filled, the connection is closed, or internally buffered data is exhausted. Regardless of whether or not the incoming data fills all the buffers, the completion indication occurs for overlapped sockets. For message-oriented sockets, an incoming message is placed into the supplied buffers, up to the total size of the buffers supplied, and the completion indication occurs for overlapped sockets. If the message is larger than the buffers supplied, the buffers are filled with the first part of the message. If the MSG_PARTIAL feature is supported by the underlying service provider, the MSG_PARTIAL flag is set in *lpFlags* and subsequent receive operation(s) will retrieve the rest of the message. If MSG_PARTIAL is not supported but the protocol is reliable, **WSARecvFrom()** generates the error WSAEMSGSIZE and a subsequent receive operation with a larger buffer can be used to retrieve the entire message. Otherwise (i.e. the protocol is unreliable and does not support MSG_PARTIAL), the excess data is lost, and **WSARecvFrom()** generates the error WSAEMSGSIZE.

lpFlags may be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *lpFlags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
MSG_PEEK	Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue. This flag is valid only for non-overlapped sockets.
MSG_OOB	Process out-of-band data (See section *3.5. Out-Of-Band data* for a discussion of this topic.)
MSG_PARTIAL	This flag is for message-oriented sockets only. On output, indicates that the data supplied is a portion of the message transmitted by the sender. Remaining portions of the message will be supplied in subsequent receive operations. A subsequent receive operation with MSG_PARTIAL flag cleared indicates end of sender's message. As an input parameter indicates that the receive operation should complete even if only part of a message has been received by the service provider.

For message-oriented sockets, the MSG_PARTIAL bit is set in the *lpFlags* parameter if a partial message is received. If a complete message is received, MSG_PARTIAL is cleared in *lpFlags*. In the case of delayed completion, the value pointed to by *lpFlags* is not updated. When completion has been indicated the application should call **WSAGetOverlappedResult()** and examine the flags pointed to by the *lpdwFlags* parameter.

Overlapped socket I/O:

If an overlapped operation completes immediately, **WSARecvFrom()** returns a value of zero and the *lpNumberOfBytesRecvd* parameter is updated with the number of bytes received and the flag bits pointed by the *lpFlags* parameter are also updated. If the

overlapped operation is successfully initiated and will complete later, **WSARecvFrom)** returns SOCKET_ERROR and indicates error code WSA_IO_PENDING. In this case, *lpNumberOfBytesRecvd* and *lpFlags* is not updated. When the overlapped operation completes the amount of data transferred is indicated either via the *cbTransferred* parameter in the completion routine (if specified), or via the *lpcbTransfer* parameter in **WSAGetOverlappedResult()**. Flag values are obtained either via the *dwFlags* parameter of the completion routine, or by examining the *lpdwFlags* parameter of **WSAGetOverlappedResult().**

This function may be called from within the completion routine of a previous **WSARecv()**, **WSARecvFrom()**, **WSASend()** or **WSASendTo()** function. For a given socket, I/O completion routines will not be nested. This permits time sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate overlapped structure. The WSAOVERLAPPED structure has the following form:

```
typedef struct _WSAOVERLAPPED {
      DWORD       Internal;         // reserved
      DWORD       InternalHigh;     // reserved
      DWORD       Offset;           // reserved
      DWORD       OffsetHigh;       // reserved
      WSAEVENT    hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* field of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use **WSAWaitForMultipleEvents()** or **WSAGetOverlappedResult()** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* field is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-NULL *lpCompletionRoutine* and later calls **WSAGetOverlappedResult()** for the same overlapped IO request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult()** to TRUE. In this case the usage of the *hEvent* field is undefined, and attempting to wait on the *hEvent* field would produce unpredictable results.

The completion routine follows the same rules as stipulated for Win32 file I/O completion routines. The completion routine will not be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents()** with the *fAlertable* parameter set to TRUE is invoked.

Transport providers allow an application to invoke send and receive operations from within the context of the socket I/O completion routine, and guarantee that, for a given socket, I/O completion routines will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The prototype of the completion routine is as follows:

void CALLBACK
CompletionRoutine(

```
    IN      DWORD               dwError,
    IN      DWORD               cbTransferred,
    IN      LPWSAOVERLAPPED     lpOverlapped,
    IN      DWORD               dwFlags
);
```

CompletionRoutine is a placeholder for an application-defined or library-defined function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes received. *dwFlags* contains information that would have appeared in *lpFlags* if the receive operation had completed immediately. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. When using **WSAWaitForMultipleEvents()**, all waiting completion routines are called before the alertable thread's wait is satisfied with a return code of WSA_IO_COMPLETION. The completion routines may be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be filled in the same order they are supplied.

Return Value

If no error occurs and the receive operation has completed immediately, **WSARecvFrom()** returns 0. Note that in this case the completion routine will have already been scheduled, and to be called once the calling thread is in the alertable state. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *lpBuffers, lpNumberOfBytesRecvd, lpFlags, lpFrom, lpFromlen, lpOverlapped, or lpCompletionRoutine* argument is not totally contained in a valid part of the user address space: the *lpFrom* buffer was too small to accommodate the peer address.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The socket has not been bound (e.g., with **bind()**).

WSAEISCONN	The socket is connected. This function is not permitted with a connected socket, whether the socket is connection-oriented or connectionless.
WSAENETRESET	The connection has been broken due to "keep-alive" activity detecting a failure while the operation was in progress.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **WSARecvFrom()** on a socket after **shutdown()** has been invoked with *how* set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	Overlapped sockets: There are too many outstanding overlapped I/O requests. Non-overlapped sockets: The socket is marked as non-blocking and the receive operation cannot be completed immediately.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and (for unreliable protocols only) any trailing portion of the message that did not fit into the buffer has been discarded.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a "hard" or "abortive" close. The application should close the socket as it is no longer useable. On a UDP datagram socket this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.
WSAEDISCON	Socket *s* is message oriented and the virtual circuit was gracefully closed by the remote side.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time
WSA_OPERATION_ABORTED	The overlapped operation has been canceled due to the closure of the socket.

See Also **WSACloseEvent(),WSACreateEvent(), WSAGetOverlappedResult(), WSASocket(), WSAWaitForMultipleEvents()**

4.46. WSAResetEvent()

Description Resets the state of the specified event object to nonsignaled.

```
#include <winsock2.h>

BOOL WSAAPI
WSAResetEvent(
        IN      WSAEVENT            hEvent
);
```

hEvent Identifies an open event object handle.

Remarks The state of the event object is set to be nonsignaled.

Return Value If the function succeeds, the return value is TRUE. If the function fails, the return value is FALSE. To get extended error information, call **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSA_INVALID_HANDLE	*hEvent* is not a valid event object handle.

See Also **WSACreateEvent(), WSASetEvent(), WSACloseEvent().**

4.47. WSASend()

Description Send data on a connected socket

```
#include <winsock2.h>

int WSAAPI
WSASend (
      IN    SOCKET                            s,
      IN    LPWSABUF                          lpBuffers,
      IN    DWORD                             dwBufferCount,
      OUT   LPDWORD                           lpNumberOfBytesSent,
      IN    DWORD                             dwFlags,
      IN    LPWSAOVERLAPPED                   lpOverlapped,
      IN    LPWSAOVERLAPPED_COMPLETION_ROUTINE
                                              lpCompletionRoutine
);
```

s A descriptor identifying a connected socket.

lpBuffers A pointer to an array of **WSABUF** structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer. This array must remain valid for the duration of the send operation.

```
typedef struct __WSABUF {
      u_long      len;  // buffer length
      char FAR *  buf;  // pointer to buffer
} WSABUF, FAR * LPWSABUF;
```

dwBufferCount The number of **WSABUF** structures in the *lpBuffers* array.

lpNumberOfBytesSent A pointer to the number of bytes sent by this call if the I/O operation completes immediately.

dwFlags Specifies the way in which the call is made.

lpOverlapped A pointer to a WSAOVERLAPPED structure (ignored for non-overlapped sockets).

lpCompletionRoutine A pointer to the completion routine called when the send operation has been completed (ignored for non-overlapped sockets).

Remarks This function provides functionality over and above the standard **send()** function in two important areas:

- It can be used in conjunction with overlapped sockets to perform overlapped send operations.
- It allows multiple send buffers to be specified making it applicable to the scatter/gather type of I/O.

WSASend() is used to write outgoing data from one or more buffers on a connection-oriented socket specified by *s*. It may also be used, however, on connectionless sockets

which have a stipulated default peer address established via the **connect()** or **WSAConnect()** function.

For overlapped sockets (created using **WSASocket()** with flag WSA_FLAG_OVERLAPPED) this will occur using overlapped I/O, unless both *lpOverlapped* and *lpCompletionRoutine* are NULL in which case the socket is treated as a non-overlapped socket. A completion indication will occur (invocation of the completion routine or setting of an event object) when the supplied buffer(s) have been consumed by the transport. If the operation does not complete immediately, the final completion status is retrieved via the completion routine or **WSAGetOverlappedResult().**

If both *lpOverlapped* and *lpCompletionRoutine* are NULL, the socket in this function will be treated as a non-overlapped socket.

For non-overlapped sockets, the last two parameters (*lpOverlapped*, *lpCompletionRoutine*) are ignored and **WSASend()** adopts the same blocking semantics as **send()**. Data is copied from the supplied buffer(s) into the transport's buffer. If the socket is non-blocking and stream-oriented, and there is not sufficient space in the transport's buffer, **WSASend()** will return with only part of the application's buffers having been consumed. Given the same buffer situation and a blocking socket, **WSASend()** will block until all of the application's buffer contents have been consumed.

The array of **WSABUF** structures pointed to by the *lpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture these **WSABUF** structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays.

For message-oriented sockets, care must be taken not to exceed the maximum message size of the underlying provider, which can be obtained by getting the value of socket option SO_MAX_MSG_SIZE. If the data is too long to pass atomically through the underlying protocol the error WSAEMSGSIZE is returned, and no data is transmitted.

Note that the successful completion of a **WSASend()** does not indicate that the data was successfully delivered.

dwFlags may be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *dwFlags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
MSG_DONTROUTE	Specifies that the data should not be subject to routing. A WinSock service provider may choose to ignore this flag.
MSG_OOB	Send out-of-band data (stream style socket such as SOCK_STREAM only).
MSG_PARTIAL	Specifies that *lpBuffers* only contains a partial message. Note that the error code WSAEOPNOTSUPP will be returned by transports which do not support partial message transmissions.

Overlapped socket I/O:
If an overlapped operation completes immediately, **WSASend()** returns a value of zero and the *lpNumberOfBytesSent* parameter is updated with the number of bytes sent. If the overlapped operation is successfully initiated and will complete later, **WSASend()** returns SOCKET_ERROR and indicates error code WSA_IO_PENDING. In this case, *lpNumberOfBytesSent* is not updated. When the overlapped operation completes the amount of data transferred is indicated either via the *cbTransferred* parameter in the completion routine (if specified), or via the *lpcbTransfer* parameter in **WSAGetOverlappedResult()**.

This function may be called from within the completion routine of a previous **WSARecv()**, **WSARecvFrom()**, **WSASend()** or **WSASendTo()** function. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate overlapped structure. The WSAOVERLAPPED structure has the following form:

```
typedef struct _WSAOVERLAPPED {
      DWORD       Internal;         // reserved
      DWORD       InternalHigh;     // reserved
      DWORD       Offset;           // reserved
      DWORD       OffsetHigh;       // reserved
      WSAEVENT    hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* field of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use **WSAWaitForMultipleEvents()** or **WSAGetOverlappedResult()** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* field is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-NULL *lpCompletionRoutine* and later calls **WSAGetOverlappedResult()** for the same overlapped IO request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult()** to TRUE. In this case the usage of the *hEvent* field is undefined, and attempting to wait on the *hEvent* field would produce unpredictable results.

The completion routine follows the same rules as stipulated for Win32 file I/O completion routines. The completion routine will not be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents()** with the *fAlertable* parameter set to TRUE is invoked.

Transport providers allow an application to invoke send and receive operations from within the context of the socket I/O completion routine, and guarantee that, for a given socket, I/O completion routines will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The prototype of the completion routine is as follows:

```
void CALLBACK
CompletionRoutine(
        IN      DWORD               dwError,
        IN      DWORD               cbTransferred,
        IN      LPWSAOVERLAPPED     lpOverlapped,
        IN      DWORD               dwFlags
);
```

CompletionRoutine is a placeholder for an application-defined or library-defined function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes sent. Currently there are no flag values defined and *dwFlags* will be zero. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. All waiting completion routines are called before the alertable thread's wait is satisfied with a return code of WSA_IO_COMPLETION. The completion routines may be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be sent in the same order they are supplied.

Return Value

If no error occurs and the send operation has completed immediately, **WSASend()** returns 0. Note that in this case the completion routine will have already been scheduled, and to be called once the calling thread is in the alertable state. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *lpBuffers, lpNumberOfBytesSent, lpOverlapped, lpCompletionRoutine* argument is not totally contained in a valid part of the user address space.

WSAENETRESET	The connection has been broken due to “keep-alive” activity detecting a failure while the operation was in progress.
WSAENOBUFS	The WinSock provider reports a buffer deadlock.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, MSG_PARTIAL is not supported, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **WSASend()** on a socket after **shutdown()** has been invoked with how set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	Overlapped sockets: There are too many outstanding overlapped I/O requests. Non-overlapped sockets: The socket is marked as non-blocking and the send operation cannot be completed immediately.
WSAEMSGSIZE	The socket is message-oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEINVAL	The socket has not been bound with **bind()**, or the socket is not created with the overlapped flag**.**
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure.
WSAECONNRESET	The virtual circuit was reset by the remote side.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	The overlapped operation has been canceled due to the closure of the socket, or the execution of the SIO_FLUSH command in **WSAIoctl()**.

See Also **WSACloseEvent(),WSACreateEvent(), WSAGetOverlappedResult(), WSASocket(), WSAWaitForMultipleEvents()**

4.48. WSASendDisconnect()

Description Initiate termination of the connection for the socket and send disconnect data.

```
#include <winsock2.h>

int WSAAPI
WSASendDisconnect (
        IN      SOCKET          s,
        IN      LPWSABUF        lpOutboundDisconnectData
);
```

s	A descriptor identifying a socket.
lpOutboundDisconnectData	A pointer to the outgoing disconnect data.

Remarks **WSASendDisconnect()** is used on connection-oriented sockets to disable transmission, and to initiate termination of the connection along with the transmission of disconnect data, if any. This is equivalent to a shutdown(SD_SEND), except that **WSASendDisconnect()** also allows sending disconnect data (in protocols that support it).

After this function has been successfully issued, subsequent sends are disallowed.

lpOutboundDisconnectData, if not NULL, points to a buffer containing the outgoing disconnect data to be sent to the remote party for retrieval by using **WSARecvDisconnect()**.

Note that **WSASendDisconnect()** does not close the socket, and resources attached to the socket will not be freed until **closesocket()** is invoked.

Comments **WSASendDisconnect()** does not block regardless of the SO_LINGER setting on the socket.

An application should not rely on being able to re-use a socket after it has been **WSASendDisconnect()**ed. In particular, a WinSock provider is not required to support the use of **connect()/WSAConnect()** on such a socket.

Return Value If no error occurs, **WSASendDisconnect()** returns 0. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAENOPROTOOPT	The parameter *lpOutboundDisconnectData* is not NULL, and the disconnect data is not supported by the service provider.

WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lpOutboundDisconnectData* argument is not totally contained in a valid part of the user address space.

See Also **connect()**, **socket()**.

4.49. WSASendTo()

Description Send data to a specific destination, using overlapped I/O where applicable.

```
#include <winsock2.h>

int WSAAPI
WSASendTo (
        IN      SOCKET                          s,
        IN      LPWSABUF                        lpBuffers,
        IN      DWORD                           dwBufferCount,
        OUT     LPDWORD                         lpNumberOfBytesSent,
        IN      DWORD                           dwFlags,
        IN      const struct sockaddr FAR *     lpTo,
        IN      int                             iToLen,
        IN      LPWSAOVERLAPPED                 lpOverlapped,
        IN      LPWSAOVERLAPPED_COMPLETION_ROUTINE
                                                lpCompletionRoutine
);
```

s A descriptor identifying a (possibly connected) socket

lpBuffers A pointer to an array of **WSABUF** structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer. This array must remain valid for the duration of the send operation.

```
typedef struct __WSABUF {
      u_long      len;  // buffer length
      char FAR *  buf;  // pointer to buffer
} WSABUF, FAR * LPWSABUF;
```

dwBufferCount The number of **WSABUF** structures in the *lpBuffers* array.

lpNumberOfBytesSent A pointer to the number of bytes sent by this call if the I/O operation completes immediately.

dwFlags Specifies the way in which the call is made.

lpTo An optional pointer to the address of the target socket.

iToLen The size of the address in *lpTo*.

lpOverlapped A pointer to a WSAOVERLAPPED structure (ignored for non-overlapped sockets).

lpCompletionRoutine A pointer to the completion routine called when the send operation has been completed (ignored for non-overlapped sockets).

Remarks This function provides functionality over and above the standard **sendto()** function in two important areas:

- It can be used in conjunction with overlapped sockets to perform overlapped send operations.
- It allows multiple send buffers to be specified making it applicable to the scatter/gather type of I/O.

WSASendTo() is normally used on a connectionless socket specified by *s* to send a datagram contained in one or more buffers to a specific peer socket identified by the *lpTo* parameter. Even if the connectionless socket has been previously **connect()**ed to a specific address, *lpTo* overrides the destination address for that particular datagram only. On a connection-oriented socket, the *lpTo* and *iToLen* parameters are ignored; in this case the **WSASendTo()** is equivalent to **WSASend()**.

For overlapped sockets (created using **WSASocket()** with flag WSA_FLAG_OVERLAPPED) this will occur using overlapped I/O, unless both *lpOverlapped* and *lpCompletionRoutine* are NULL in which case the socket is treated as a non-overlapped socket. A completion indication will occur (invocation of the completion routine or setting of an event object) when the supplied buffer(s) have been consumed by the transport. If the operation does not complete immediately, the final completion status is retrieved via the completion routine or **WSAGetOverlappedResult().**

If both *lpOverlapped* and *lpCompletionRoutine* are NULL, the socket in this function will be treated as a non-overlapped socket.

For non-overlapped sockets, the last two parameters (*lpOverlapped, lpCompletionRoutine*) are ignored and **WSASendTo()** adopts the same blocking semantics as **send()**. Data is copied from the supplied buffer(s) into the transport's buffer. If the socket is non-blocking and stream-oriented, and there is not sufficient space in the transport's buffer, **WSASendTo()** will return with only part of the application's buffers having been consumed. Given the same buffer situation and a blocking socket, **WSASendTo()** will block until all of the application's buffer contents have been consumed.

The array of **WSABUF** structures pointed to by the *lpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture these **WSABUF** structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays.

For message-oriented sockets, care must be taken not to exceed the maximum message size of the underlying transport, which can be obtained by getting the value of socket option SO_MAX_MSG_SIZE. If the data is too long to pass atomically through the underlying protocol the error WSAEMSGSIZE is returned, and no data is transmitted.

Note that the successful completion of a **WSASendTo()** does not indicate that the data was successfully delivered.

dwFlags may be used to influence the behavior of the function invocation beyond the options specified for the associated socket. That is, the semantics of this function are determined by the socket options and the *dwFlags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning

MSG_DONTROUTE
Specifies that the data should not be subject to routing. A WinSock service provider may choose to ignore this flag.

MSG_OOB Send out-of-band data (stream style socket such as SOCK_STREAM only).

MSG_PARTIAL Specifies that *lpBuffers* only contains a partial message. Note that the error code WSAEOPNOTSUPP will be returned by transports which do not support partial message transmissions.

Overlapped socket I/O:
If an overlapped operation completes immediately, **WSASendTo()** returns a value of zero and the *lpNumberOfBytesSent* parameter is updated with the number of bytes sent. If the overlapped operation is successfully initiated and will complete later, **WSASendTo()** returns SOCKET_ERROR and indicates error code WSA_IO_PENDING. In this case, *lpNumberOfBytesSent* is not updated. When the overlapped operation completes the amount of data transferred is indicated either via the *cbTransferred* parameter in the completion routine (if specified), or via the *lpcbTransfer* parameter in **WSAGetOverlappedResult()**.

This function may be called from within the completion routine of a previous **WSARecv()**, **WSARecvFrom()**, **WSASend()** or **WSASendTo()** function. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The *lpOverlapped* parameter must be valid for the duration of the overlapped operation. If multiple I/O operations are simultaneously outstanding, each must reference a separate overlapped structure. The WSAOVERLAPPED structure has the following form:

```
typedef struct _WSAOVERLAPPED {
      DWORD       Internal;         // reserved
      DWORD       InternalHigh;     // reserved
      DWORD       Offset;           // reserved
      DWORD       OffsetHigh;       // reserved
      WSAEVENT    hEvent;
} WSAOVERLAPPED, FAR * LPWSAOVERLAPPED;
```

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* field of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use **WSAWaitForMultipleEvents()** or **WSAGetOverlappedResult()** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* field is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-NULL *lpCompletionRoutine* and later calls **WSAGetOverlappedResult()** for the same overlapped IO request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult()** to TRUE. In this case the usage of the *hEvent* field is undefined, and attempting to wait on the *hEvent* field would produce unpredictable results.

The completion routine follows the same rules as stipulated for Win32 file I/O completion routines. The completion routine will not be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents()** with the *fAlertable* parameter set to TRUE is invoked.

Transport providers allow an application to invoke send and receive operations from within the context of the socket I/O completion routine, and guarantee that, for a given socket, I/O completion routines will not be nested. This permits time-sensitive data transmissions to occur entirely within a preemptive context.

The prototype of the completion routine is as follows:

```
void CALLBACK
CompletionRoutine(
        IN      DWORD               dwError,
        IN      DWORD               cbTransferred,
        IN      LPWSAOVERLAPPED     lpOverlapped,
        IN      DWORD               dwFlags
);
```

CompletionRoutine is a placeholder for an application-defined or library-defined function name. *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. *cbTransferred* specifies the number of bytes sent. Currently there are no flag values defined and *dwFlags* will be zero. This function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. All waiting completion routines are called before the alertable thread's wait is satisfied with a return code of WSA_IO_COMPLETION. The completion routines may be called in any order, not necessarily in the same order the overlapped operations are completed. However, the posted buffers are guaranteed to be sent in the same order they are supplied.

Return Value

If no error occurs and the send operation has completed immediately, **WSASendTo()** returns 0. Note that in this case the completion routine will have already been scheduled, and to be called once the calling thread is in the alertable state. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code may be retrieved by calling **WSAGetLastError()**. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion will be indicated at a later time. Any other error code indicates that the overlapped operation was not successfully initiated and no completion indication will occur.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set.

WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *lpBuffers, lpTo, lpNumberOfBytesSent, lpOverlapped, or lpCompletionRoutine* parameters are not part of the user address space, or the *lpTo* argument is too small.
WSAENETRESET	The connection has been broken due to "keep-alive" activity detecting a failure while the operation was in progress.
WSAENOBUFS	The WinSock provider reports a buffer deadlock.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only)
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, MSG_PARTIAL is not supported, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shutdown; it is not possible to **WSASendTo()** on a socket after **shutdown()** has been invoked with how set to SD_SEND or SD_BOTH.
WSAEWOULDBLOCK	Overlapped sockets: There are too many outstanding overlapped I/O requests. Non-overlapped sockets: The socket is marked as non-blocking and the send operation cannot be completed immediately.
WSAEMSGSIZE	The socket is message-oriented, and the message is larger than the maximum supported by the underlying transport.
WSAEINVAL	The socket has not been bound with **bind()**, or the socket is not created with the overlapped flag**.**
WSAECONNABORTED	The virtual circuit was aborted due to timeout or other failure.
WSAECONNRESET	The virtual circuit was reset by the remote side.

WSAEADDRNOTAVAIL	The remote address is not a valid address (e.g., ADDR_ANY).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAEDESTADDRREQ	A destination address is required.
WSAENETUNREACH	The network can't be reached from this host at this time.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	The overlapped operation has been canceled due to the closure of the socket, or the execution of the SIO_FLUSH command in **WSAIoctl()**.

See Also **WSACloseEvent(),WSACreateEvent(), WSAGetOverlappedResult(), WSASocket(), WSAWaitForMultipleEvents()**

4.50. WSASetBlockingHook()

Description Establish an application-supplied blocking hook function. **WSASetBlockingHook()** is only available for WinSock 1.1 apps (that is, if at least one thread within the process negotiates version 1.0 or 1.1 at **WSAStartup()**).

Important Note: *This function is for backwards compatibility with WinSock 1.1 and is not considered part of the WinSock 2 specification. WinSock 2 applications should* ***not*** *use this function.*

```
#include <winsock2.h>

FARPROC WSAAPI
WSASetBlockingHook (
        IN      FARPROC             lpBlockFunc
);
```

lpBlockFunc A pointer to the procedure instance address of the blocking function to be installed.

Remarks This function installs a new function which a WinSock implementation should use to implement blocking socket function calls.

A WinSock implementation includes a default mechanism by which blocking socket functions are implemented. The function **WSASetBlockingHook()** gives the application the ability to execute its own function at "blocking" time in place of the default function. Note: the application blocking hook function must be exported.

When an application invokes a blocking WinSock operation, WinSock initiates the operation and then enters a loop which is similar to the following pseudo code:

```
for(;;) {
     /* Look for WinSock implementation's messages (only */
     /* necessary if WinSock uses messages internally)   */
     if (PeekMessage(&msg,hMyWnd,MYFIRST,MYLAST,PM_REMOVE) {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }

     /* flush messages for good user response */
     BlockingHook();

     /* check for WSACancelBlockingCall() */
     if(operation_cancelled())
          break;
     /* check to see if operation completed */
     if(operation_complete())
          break;     /* normal completion */
}
```

Note that WinSock implementations may perform the above steps in a different order; for example, the check for operation complete may occur before calling the blocking hook. The default **BlockingHook()** function is equivalent to:

```
BOOL DefaultBlockingHook(void) {
     MSG msg;
     BOOL ret;
     /* get the next message if any */
     ret = (BOOL)PeekMessage(&msg,NULL,0,0,PM_REMOVE);
     /* if we got one, process it */
     if (ret) {/* TRUE if we got a message */
          TranslateMessage(&msg);
          DispatchMessage(&msg);
     }

     return ret;
}
```

The **WSASetBlockingHook()** function is provided to support those applications which require more complex message processing - for example, those employing the MDI (multiple document interface) model. It is not intended as a mechanism for performing general applications functions. In particular, the only WinSock function which may be issued from a custom blocking hook function is **WSACancelBlockingCall()**, which will cause the blocking loop to terminate.

This function is implemented on a per-task basis for non-multithreaded versions of Windows and on a per-thread basis for multithreaded versions of Windows such as Windows NT. It thus provides for a particular task or thread to replace the blocking mechanism without affecting other tasks or threads.

In multithreaded versions of Windows, there is no default blocking hook--blocking calls block the thread that makes the call. However, an application may install a specific blocking hook by calling **WSASetBlockingHook().**
This allows easy portability of applications that depend on the blocking hook behavior.

Return Value

The return value is a pointer to the procedure-instance of the previously installed blocking function. The application or library that calls the **WSASetBlockingHook ()** function should save this return value so that it can be restored if necessary. (If "nesting" is not important, the application may simply discard the value returned by **WSASetBlockingHook()** and eventually use **WSAUnhookBlockingHook()** to restore the default mechanism.) If the operation fails, a NULL pointer is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	*lpBlockFunc* is not in a valid part of the process address space.
WSAEOPNOTSUPP	The caller is not a WinSock 1.0 or 1.1 client.

See Also **WSAUnhookBlockingHook(), WSAIsBlocking(), WSACancelBlockingCall()**

4.51. WSASetEvent()

Description Sets the state of the specified event object to signaled.

```
#include <winsock2.h>

BOOL WSAAPI
WSASetEvent(
        IN      WSAEVENT        hEvent
);
```

hEvent Identifies an open event object handle.

Remarks The state of the event object is set to be signaled.

Return Value If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSA_INVALID_HANDLE	*hEvent* is not a valid event object handle.

See Also **WSACreateEvent(), WSAResetEvent(), WSACloseEvent().**

4.52. WSASetLastError()

Description Set the error code which can be retrieved by **WSAGetLastError()**.

```
#include <winsock2.h>

void WSAAPI
WSASetLastError (
        IN      int             iError
);
```

iError Specifies the error code to be returned by a subsequent **WSAGetLastError()** call.

Remarks This function allows an application to set the error code to be returned by a subsequent **WSAGetLastError()** call for the current thread. Note that any subsequent WinSock routine called by the application will override the error code as set by this routine.

The error code set by **WSASetLastError()** is different from the error code reset by **getsockopt()** SO_ERROR.

Return Value None.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.

See Also **WSAGetLastError(), getsockopt()**

4.53. WSASocket()

Description Create a socket which is bound to a specific transport service provider, optionally create and/or join a socket group.

```
#include <winsock2.h>

SOCKET WSAAPI
WSASocket (
        IN      int                     af,
        IN      int                     type,
        IN      int                     protocol,
        IN      LPWSAPROTOCOL_INFO      lpProtocolInfo,
        IN      GROUP                   g,
        IN      DWORD                   dwFlags
);
```

af	An address family specification.
type	A type specification for the new socket.
protocol	A particular protocol to be used with the socket which is specific to the indicated address family.
lpProtocolInfo	A pointer to a WSAPROTOCOL_INFO struct that defines the characteristics of the socket to be created.
g	Reserved for future use with socket groups: The identifier of the socket group.
dwFlags	The socket attribute specification.

Remarks This function causes a socket descriptor and any related resources to be allocated and associated with a transport service provider. By default, the created socket will **not** have the overlapped attribute. If *lpProtocolInfo* is NULL, the WinSock 2 DLL uses the first three parameters (*af, type, protocol*) to determine which service provider is used by selecting the first transport provider able to support the stipulated address family, socket type and protocol values. If the *lpProtocolInfo* is not NULL, the socket will be bound to the provider associated with the indicated WSAPROTOCOL_INFO struct. In this instance, the application may supply the manifest constant FROM_PROTOCOL_INFO as the value for any of *af, type* or *protocol*. This indicates that the corresponding values from the indicated WSAPROTOCOL_INFO struct (iAddressFamily, iSocketType, iProtocol) are to be assumed. In any case, the values supplied for *af, type* and *protocol* are supplied unmodified to the transport service provider via the corresponding parameters to the **WSPSocket()** function in the SPI.

When selecting a protocol and its supporting service provider based on *af, type* and *protocol* this procedure will only choose a base protocol or a protocol chain, not a protocol layer by itself. Unchained protocol layers are not considered to have "partial matches" on *type* or *af* either. That is, they do not lead to an error code of WSAEAFNOSUPPORT or WSAEPROTONOSUPPORT if no suitable protocol is found.

Note: the manifest constant AF_UNSPEC continues to be defined in the header file but its use is **strongly discouraged**, as this may cause ambiguity in interpreting the value of the *protocol* parameter.

Reserved for future use with socket groups: Parameter *g* is used to indicate the appropriate actions on socket groups:

if *g* is an existing socket group ID, join the new socket to this group, provided all the requirements set by this group are met; or

if *g* = SG_UNCONSTRAINED_GROUP, create an unconstrained socket group and have the new socket be the first member; or

if *g* = SG_CONSTRAINED_GROUP, create a constrained socket group and have the new socket be the first member; or

if *g* = zero, no group operation is performed

For unconstrained groups, any set of sockets may be grouped together as long as they are supported by a single service provider. A constrained socket group may consist only of connection-oriented sockets, and requires that connections on all grouped sockets be to the same address on the same host. For newly created socket groups, the new group ID can be retrieved by using **getsockopt()** with option SO_GROUP_ID, if this operation completes successfully. A socket group and its associated ID remain valid until the last socket belonging to this socket group is closed. Socket group IDs are unique across all processes for a given service provider.

The *dwFlags* parameter may be used to specify the attributes of the socket by or-ing any of the following Flags:

Flag	Meaning
WSA_FLAG_OVERLAPPED	This flag causes an overlapped socket to be created. Overlapped sockets may utilize **WSASend()**, **WSASendTo()**, **WSARecv()**, **WSARecvFrom()** and **WSAIoctl()** for overlapped I/O operations, which allows multiple these operations to be initiated and in progress simultaneously. All functions that allow overlapped operation (**WSASend(), WSARecv(),WSASendTo(), WSARecvFrom(), WSAIoctl()**) also support non-overlapped usage on an overlapped socket if the values for parameters related to overlapped operation are NULL.
WSA_FLAG_MULTIPOINT_C_ROOT	Indicates that the socket created will be a c_root in a multipoint session. Only allowed if a rooted control plane is indicated in the protocol's WSAPROTOCOL_INFO struct. Refer to Appendix B. Multipoint and Multicast Semantics for additional information.
WSA_FLAG_MULTIPOINT_C_LEAF	Indicates that the socket created will be a c_leaf in a multicast session. Only allowed if XP1_SUPPORT_MULTIPOINT is indicated in the protocol's WSAPROTOCOL_INFO struct. Refer to Appendix B. Multipoint and Multicast Semantics for additional information.
WSA_FLAG_MULTIPOINT_D_ROOT	Indicates that the socket created will be a d_root in a multipoint session. Only allowed if a rooted data plane is indicated in the protocol's WSAPROTOCOL_INFO struct. Refer to Appendix B. Multipoint and Multicast Semantics for additional information.

WSA_FLAG_MULTIPOINT_D_LEAF
: Indicates that the socket created will be a d_leaf in a multipoint session. Only allowed if XP1_SUPPORT_MULTIPOINT is indicated in the protocol's WSAPROTOCOL_INFO struct. Refer to Appendix B. Multipoint and Multicast Semantics for additional information.

Important note: for multipoint sockets, exactly one of WSA_FLAG_MULTIPOINT_C_ROOT or WSA_FLAG_MULTIPOINT_C_LEAF **must** be specified, and exactly one of WSA_FLAG_MULTIPOINT_D_ROOT or WSA_FLAG_MULTIPOINT_D_LEAF **must** be specified. Refer to Appendix B. Multipoint and Multicast Semantics for additional information.

Connection-oriented sockets such as SOCK_STREAM provide full-duplex connections, and must be in a connected state before any data may be sent or received on them. A connection to another socket is created with a **connect()**/**WSAConnect()** call. Once connected, data may be transferred using **send()**/**WSASend()** and **recv()**/**WSARecv()** calls. When a session has been completed, a **closesocket()** must be performed.

The communications protocols used to implement a reliable, connection-oriented socket ensure that data is not lost or duplicated. If data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, the connection is considered broken and subsequent calls will fail with the error code set to WSAETIMEDOUT.

Connectionless, message-oriented sockets allow sending and receiving of datagrams to and from arbitrary peers using **sendto()/WSASendTo()** and **recvfrom()/WSARecvFrom()**. If such a socket is **connect()**ed to a specific peer, datagrams may be sent to that peer using **send()**/**WSASend()** and may be received from (only) this peer using **recv()**/**WSARecv()**.

Support for sockets with type RAW is not required but service providers are encouraged to support raw sockets whenever it makes sense to do so.

Shared Sockets When a special WSAPROTOCOL_INFO struct (obtained via the **WSADuplicateSocket()** function and used to create additional descriptors for a shared socket) is passed as an input parameter to **WSASocket(),** the *g* and *dwFlags* parameters are **ignored.** Such a WSAPROTOCOL_INFO struct may only be used once, otherwise the error code WSAEINVAL will result.

Return Value If no error occurs, **WSASocket()** returns a descriptor referencing the new socket. Otherwise, a value of INVALID_SOCKET is returned, and a specific error code may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEAFNOSUPPORT	The specified address family is not supported.

WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEMFILE	No more socket descriptors are available.
WSAENOBUFS	No buffer space is available. The socket cannot be created.
WSAEPROTONOSUPPORT	The specified protocol is not supported
WSAEPROTOTYPE	The specified protocol is the wrong type for this socket.
WSAESOCKTNOSUPPORT	The specified socket type is not supported in this address family.
WSAEINVAL	The parameter *g* specified is not valid, or the WSAPROTOCOL_INFO structure that lpProtocolInfo points to is incomplete, the contents are invalid or the WSAPROTOCOL_INFO struct has already been used in an earlier duplicate socket operation.
WSAEFAULT	*lpProtocolInfo* argument is not in a valid part of the process address space.
WSAINVALIDPROVIDER	The service provider returned a version other than 2.2
WSAINVALIDPROCTABLE	The service provider returned an invalid or incomplete procedure table to the WSPStartup

See Also **accept()**, **bind()**, **connect()**, **getsockname()**, **getsockopt()**, **setsockopt()**, **listen()**, r**ecv()**, **recvfrom()**, **select()**, **send()**, **sendto()**, **shutdown()**, **ioctlsocket()**.

4.54. WSAStartup()

Description Initiate use of the WinSock DLL by a process.

```
#include <winsock2.h>

int WSAAPI
WSAStartup (
        IN      WORD            wVersionRequested,
        OUT     LPWSADATA       lpWSAData
);
```

wVersionRequested	The highest version of WinSock support that the caller can use. The high order byte specifies the minor version (revision) number; the low-order byte specifies the major version number.
lpWSAData	A pointer to the **WSADATA** data structure that is to receive details of the WinSock implementation.

Remarks This function **MUST** be the first WinSock function called by an application or DLL. It allows an application or DLL to specify the version of WinSock required and to retrieve details of the specific WinSock implementation. The application or DLL may only issue further WinSock functions after a successful **WSAStartup()** invocation.

In order to support future WinSock implementations and applications which may have functionality differences from current version of WinSock, a negotiation takes place in **WSAStartup()**. The caller of **WSAStartup()** and the WinSock DLL indicate to each other the highest version that they can support, and each confirms that the other's highest version is acceptable. Upon entry to **WSAStartup()**, the WinSock DLL examines the version requested by the application. If this version is equal to or higher than the lowest version supported by the DLL, the call succeeds and the DLL returns in *wHighVersion* the highest version it supports and in *wVersion* the minimum of its high version and *wVersionRequested.* The WinSock DLL then assumes that the application will use *wVersion.* If the *wVersion* field of the **WSADATA** structure is unacceptable to the caller, it should call **WSACleanup()** and either search for another WinSock DLL or fail to initialize.

Note that it is legal and possible for an application written to this version of the specification to successfully negotiate a higher version number than the version of this specification. In such a case, the application is only guaranteed access to higher-version functionality that fits within the syntax defined in this version, such as new Ioctl codes and new behavior of existing functions. New functions, for example, may be inaccessible. To be guaranteed full access to new syntax of a future version, the application must fully conform to that future version, such as compiling against a new header file, linking to a new library, etc.

This negotiation allows both a WinSock DLL and a WinSock application to support a range of WinSock versions. An application can successfully utilize a WinSock DLL if there is any overlap in the version ranges. The following chart gives examples of how **WSAStartup()** works in conjunction with different application and WinSock DLL versions:

App versions	DLL Versions	*wVersionRequested*	*wVersion*	*wHighVersion*	End Result
1.1	1.1	1.1	1.1	1.1	use 1.1
1.0 1.1	1.0	1.1	1.0	1.0	use 1.0
1.0	1.0 1.1	1.0	1.0	1.1	use 1.0
1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1	1.0	1.1	1.0	1.0	Application fails
1.0	1.1	1.0	---	---	WSAVERNOTSUPPORTED
1.0 1.1	1.0 1.1	1.1	1.1	1.1	use 1.1
1.1 2.0	1.1	2.0	1.1	1.1	use 1.1
2.0	2.0	2.0	2.0	2.0	use 2.0

The following code fragment demonstrates how an application which supports only version 2.2 of WinSock makes a **WSAStartup()** call:

```
WORD wVersionRequested;
WSADATA wsaData;
int err;

wVersionRequested = MAKEWORD( 2, 2 );

err = WSAStartup( wVersionRequested, &wsaData );
if ( err != 0 ) {
    /* Tell the user that we couldn't find a useable */
    /* WinSock DLL.                                  */
    return;
}

/* Confirm that the WinSock DLL supports 2.2.*/
/* Note that if the DLL supports versions greater    */
/* than 2.2 in addition to 2.2, it will still return */
/* 2.2 in wVersion since that is the version we      */
/* requested.                                        */

if ( LOBYTE( wsaData.wVersion ) != 2 ||
        HIBYTE( wsaData.wVersion ) != 2 ) {
    /* Tell the user that we couldn't find a useable */
    /* WinSock DLL.                                  */
    WSACleanup( );
    return;
}

/* The WinSock DLL is acceptable.  Proceed.  */
```

Once an application or DLL has made a successful **WSAStartup()** call, it may proceed to make other WinSock calls as needed. When it has finished using the services of the WinSock DLL, the application or DLL must call **WSACleanup()** in order to allow the WinSock DLL to free any resources for the application.

Details of the actual WinSock implementation are described in the WSAData structure defined as follows:

```
struct WSAData {
      WORD              wVersion;
      WORD              wHighVersion;
      char              szDescription[WSADESCRIPTION_LEN+1];
      char              szSystemStatus[WSASYSSTATUS_LEN+1];
      unsigned short    iMaxSockets;
```

```
      unsigned short    iMaxUdpDg;
      char FAR *        lpVendorInfo;
};
```

The members of this structure are:

Element	Usage
wVersion	The version of the WinSock specification that the WinSock DLL expects the caller to use.
wHighVersion	The highest version of the WinSock specification that this DLL can support (also encoded as above). Normally this will be the same as *wVersion*.
szDescription	A null-terminated ASCII string into which the WinSock DLL copies a description of the WinSock implementation. The text (up to 256 characters in length) may contain any characters except control and formatting characters: the most likely use that an application will put this to is to display it (possibly truncated) in a status message.
szSystemStatus	A null-terminated ASCII string into which the WinSock DLL copies relevant status or configuration information. The WinSock DLL should use this field only if the information might be useful to the user or support staff: it should not be considered as an extension of the *szDescription* field.
iMaxSockets	This field is retained for backwards compatibility, ***but should be ignored for version 2.0 and onward*** as no single value can be appropriate for all underlying service providers.
iMaxUdpDg	***This value should be ignored for version 2.0 and onward***. It is retained for compatibility with Windows Sockets specification 1.1, but should not be used when developing new applications. For the actual maximum message size specific to a particular WinSock service provider and socket type, applications should use **getsockopt()** to retrieve the value of option SO_MAX_MSG_SIZE after a socket has been created.
lpVendorInfo	***This value should be ignored for version 2.0 and onward***. It is retained for compatibility with Windows Sockets specification 1.1. Applications needing to access vendor-specific configuration information should use **getsockopt()** to retrieve the value of option PVD_CONFIG. The definition of this value (if utilized) is beyond the scope of this specification.

Note that an application should ignore the *iMaxsockets*, *iMaxUdpDg*, and *lpVendorInfo* fields in WSAData if the value in *wVersion* after a successful call to **WSAStartup()** is at least 2.0. This is because the architecture of WinSock has been changed in version 2.0 to support multiple providers, and WSAData no longer applies to a single vendor's stack. Two new socket options are introduced to supply provider-specific information: SO_MAX_MSG_SIZE (replaces the *iMaxUdpDg* element) and PVD_CONFIG (allows any other provider-specific configuration to occur).

An application or DLL may call **WSAStartup()** more than once if it needs to obtain the WSAData structure information more than once. On each such call the application may specify any version number supported by the DLL.

There must be one **WSACleanup()** call corresponding to every successful **WSAStartup()** call to allow third-party DLLs to make use of a WinSock DLL on behalf of an application. This means, for example, that if an application calls **WSAStartup()** three times, it must call **WSACleanup()** three times. The first two calls

to **WSACleanup()** do nothing except decrement an internal counter; the final **WSACleanup()** call for the task does all necessary resource deallocation for the task.

Return Value

WSAStartup() returns zero if successful. Otherwise it returns one of the error codes listed below. Note that the normal mechanism whereby the application calls **WSAGetLastError()** to determine the error code cannot be used, since the WinSock DLL may not have established the client data area where the "last error" information is stored.

Error Codes

WSASYSNOTREADY	Indicates that the underlying network subsystem is not ready for network communication.
WSAVERNOTSUPPORTED	The version of WinSock support requested is not provided by this particular WinSock implementation.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAEPROCLIM	Limit on the number of tasks supported by the Windows Sockets implementation has been reached.
WSAEFAULT	The *lpWSAData* is not a valid pointer.

See Also

send(), **sendto()**, **WSACleanup()**

4.55. WSAUnhookBlockingHook()

Description Restore the default blocking hook function. **WSAUnhookBlockingHook()** is only available for WinSock 1.1 apps (that is, if at least one thread within the process negotiates version 1.0 or 1.1 at **WSAStartup()**).

Important Note: *This function is for backwards compatibility with WinSock 1.1 and is not considered part of the WinSock 2 specification. WinSock 2 applications should* ***not*** *use this function.*

#include <winsock2.h>

int WSAAPI WSAUnhookBlockingHook (void);

Remarks This function removes any previous blocking hook that has been installed and reinstalls the default blocking mechanism.

WSAUnhookBlockingHook() will always install the default mechanism, not the previous mechanism. If an application wish to nest blocking hooks - i.e. to establish a temporary blocking hook function and then revert to the previous mechanism (whether the default or one established by an earlier **WSASetBlockingHook()**) - it must save and restore the value returned by **WSASetBlockingHook()**; it cannot use **WSAUnhookBlockingHook()**.

In multithreaded versions of Windows such as Windows NT, there is no default blocking hook. Calling **WSAUnhookBlockingHook()** disables any blocking hook installed by the application and any blocking calls made block the thread which made the call.

Return Value The return value is 0 if the operation was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAEINPROGRESS	A blocking Windows Sockets operation is in progress.
WSAEOPNOTSUPP	The caller is not a WinSock 1.0 or 1.1 client.

See Also **WSASetBlockingHook(), WSAIsBlocking(), WSACancelBlockingCall()**

4.56. WSAWaitForMultipleEvents()

Description Returns either when any one or all of the specified event objects are in the signaled state, or when the timeout interval expires.

```
#include <winsock2.h>

DWORD WSAAPI
WSAWaitForMultipleEvents(
    IN    DWORD                 cEvents,
    IN    const WSAEVENT FAR *  lphEvents,
    IN    BOOL                  fWaitAll,
    IN    DWORD                 dwTimeout,
    IN    BOOL                  fAlertable
);
```

cEvents Specifies the number of event object handles in the array pointed to by *lphEvents*. The maximum number of event object handles is WSA_MAXIMUM_WAIT_EVENTS. One or more events must be specified.

lphEvents Points to an array of event object handles.

fWaitAll Specifies the wait type. If TRUE, the function returns when all event objects in the *lphEvents* array are signaled at the same time. If FALSE, the function returns when any one of the event objects is signaled. In the latter case, the return value indicates the event object whose state caused the function to return.

dwTimeout Specifies the time-out interval, in milliseconds. The function returns if the interval expires, even if conditions specified by the *fWaitAll* parameter are not satisfied. If *dwTimeout* is zero, the function tests the state of the specified event objects and returns immediately. If *dwTimeout* is WSA_INFINITE, the function's time-out interval never expires.

fAlertable Specifies whether the function returns when the system queues an I/O completion routine for execution by the calling thread. If TRUE, the completion routine is executed and the function returns. If FALSE, the completion routine is not executed when the function returns.

Remarks **WSAWaitForMultipleEvents()** returns either when any one or when all of the specified objects are in the signaled state, or when the time-out interval elapses. This function is also used to perform an alertable wait by setting the parameter *fAltertable* to be TRUE. This enables the function to return when the system queues an I/O completion routine to be executed by the calling thread.

When *fWaitAll* is TRUE, the function's wait condition is satisfied only when the state of all objects is signaled at the same time. The function does not modify the state of the specified objects until all objects are simultaneously signaled.

Applications that simply need to enter an alertable wait state without waiting for any event objects to be signaled should use the Win32 **SleepEx()** function.

Return Value If the function succeeds, the return value indicates the event object that caused the function to return.

If the function fails, the return value is WSA_WAIT_FAILED. To get extended error information, call **WSAGetLastError()**.

The return value upon success is one of the following values:

Value	Meaning
WSA_WAIT_EVENT_0 to (WSA_WAIT_EVENT_0 + cEvents - 1)	If *fWaitAll* is TRUE, the return value indicates that the state of all specified event objects is signaled. If *fWaitAll* is FALSE, the return value minus WSA_WAIT_EVENT_0 indicates the *lphEvents* array index of the object that satisfied the wait.
WAIT_IO_COMPLETION	One or more I/O completion routines are queued for execution.
WSA_WAIT_TIMEOUT	The time-out interval elapsed and the conditions specified by the *fWaitAll* parameter are not satisfied.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to complete the operation.
WSA_INVALID_HANDLE	One or more of the values in the *lphEvents* array is not a valid event object handle.
WSA_INVALID_PARAMETER	The *cEvents* parameter does not contain a valid handle count.

See Also **WSACreateEvent(), WSACloseEvent().**

4.57. WSAProviderConfigChange()

Description Notifies the application when the provider configuration is changed

```
#include <winsock2.h>

int WSAAPI
WSAProviderConfigChange(
        IN OUT LPHANDLE                         lpNotificationHandle,
        IN      LPWSAOVERLAPPED                 lpOverlapped,
        IN      LPWSAOVERLAPPED_COMPLETION_ROUTINE
                                                lpCompletionRoutine
);
```

lpNotificationHandle	A pointer to notification handle; if the notification handle is set to NULL (the handle value not the pointer itself), this function returns notification handle in the location pointed by *lpNotificationHandle*
lpOverlapped	A pointer to a WSAOVERLAPPED structure.
lpCompletionRoutine	A pointer to the completion routine called when the provider change notification is received

Remarks **WSAProviderConfigChange ()** notifies the application of provider (both transport and name space) installation or removal in Win32 operating environments that support such configuration change without requiring a restart. When called for the first time (*lpNotificationHandle* parameter points to NULL handle), this function completes immediately and returns notification handle in the location pointed by *lpNotificationHandle* that can be used in subsequent calls to receive notifications of provider installation and removal. The second and any subsequent calls only complete when provider information changes since the time the call was made It is expected (but not required) that that application uses overlapped IO on second and subsequent calls to **WSAProviderConfigChange()**, in which case the call will return immediately and application will be notified of provider configuration changes using the completion mechanism chosen through specified overlapped completion parameters.

Notification handle returned by **WSAProtocolConfigChange()** is like any regular operating system handle that should be closed (when no longer needed) using Win32 **CloseHandle()** call.

The following sequence of actions can be used to guarantee that application always has current protocol configuration information:

- call WSAProviderConfigChange
- call WSAEnumProtocols and/or WSAEnumNameSpaces
- whenever WSAProtocolConfigChange notifies application of provider configuration change (via blocking or overlapped IO), the whole sequence of actions should be repeated

Return Value If no error occurs the **WSAProviderConfigChange()** returns 0. Otherwise, a value of SOCKET_ERROR is returned and a specific error code may be retrieved by calling **WSAGetLastError()**. The error code WSA_IO_PENDING indicates that the overlapped operation has been successfully initiated and that completion (and thus change event) will be indicated at a later time

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to complete the operation.
WSA_INVALID_HANDLE	Value pointed by *lpNotificationHandle* parameter is not a valid notification handle.
WSAEOPNOTSUPP	Current operating system environment does not support provider installation/removal without restart..

5. Name Resolution and Registration

WinSock 2 includes a new set of API functions that standardize the way applications access and use the various network naming services. When using these new functions, WinSock 2 applications need not be cognizant of the widely differing protocols associated with name services such as DNS, NIS, X.500, SAP, etc. To maintain full backwards compatibility with WinSock 1.1, all of the existing **getXbyY()** and asynchronous **WSAAsyncGetXbyY()** database lookup functions continue to be supported, but are implemented in the WinSock service provider interface in terms of the new name resolution capabilities. See section *4.3.4 getservbyname() and getservbyport.*.

5.1. Protocol-Independent Name Resolution

In developing a protocol-independent client/server application, there are two basic requirements that exist with respect to name resolution and registration:

- The ability of the server half of the application (hereafter referred to as a service) to register its existence within (or become accessible to) one or more name spaces
- The ability of the client application to find the service within a name space and obtain the required transport protocol and addressing information

For those accustomed to developing TCP/IP based applications, this may seem to involve little more than looking up a host address and then using an agreed upon port number. Other networking schemes, however, allow the location of the service, the protocol used for the service, and other attributes to be discovered at run time. To accommodate the broad diversity of capabilities found in existing name services, the WinSock 2 interface adopts the model described below.

5.1.1. Name Resolution Model

A *name space* refers to some capability to associate (as a minimum) the protocol and addressing attributes of a network service with one or more human-friendly names. Many name spaces are currently in wide use including the Internet's Domain Name System(DNS), the bindery and Netware Directory Services (NDS) from Novell, X.500, etc. These name spaces vary widely in how they are organized and implemented. Some of their properties are particularly important to understand from the perspective of WinSock name resolution.

5.1.1.1. Types of Name Spaces

There are three different types of name spaces in which a service could be registered:

- dynamic
- static
- persistent

Dynamic name spaces allow services to register with the name space on the fly, and for clients to discover the available services at run time. Dynamic name spaces frequently rely on broadcasts to indicate the continued availability of a network service. Examples of dynamic name spaces include the SAP name space used within a Netware environment and the NBP name space used by Appletalk.

Static name spaces require all of the services to be registered ahead of time, i.e. when the name space is created. The DNS is an example of a static name space. Although there is a programmatic way to resolve names, there is no programmatic way to register names.

Persistent name spaces allow services to register with the name space on the fly. Unlike dynamic name spaces however, persistent name spaces retain the registration information in non-volatile storage where it remains until such time as the service requests that it be removed. Persistent name spaces are typified by directory services such as X.500 and the NDS (Netware Directory Service). These environments

allow the adding, deleting, and modification of service properties. In addition, the service object representing the service within the directory service could have a variety of attributes associated with the service. The most important attribute for client applications is the service's addressing information.

5.1.1.2. Name Space Organization

Many name spaces are arranged hierarchically. Some, such as X.500 and NDS, allow unlimited nesting. Others allow services to be combined into a single level of hierarchy or "group." This is typically referred to as a workgroup. When constructing a query, it is often necessary to establish a context point within a name space hierarchy from which the search will begin.

5.1.1.3. Name Space Provider Architecture

Naturally, the programmatic interfaces used to query the various types of name spaces and to register information within a name space (if supported) differ widely. A *name space provider* is a locally-resident piece of software that knows how to map between WinSock's name space SPI and some existing name space (which could be implemented locally or be accessed via the network). This is illustrated as follows:

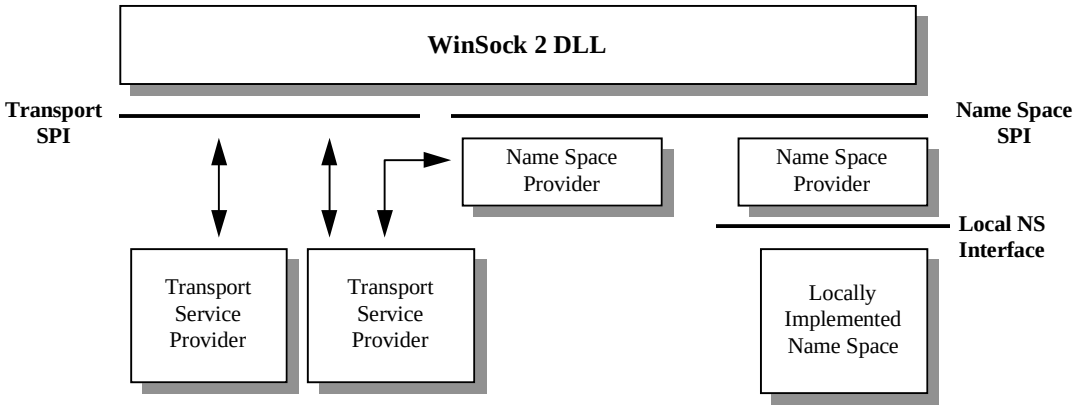

Figure 4 Name Space Provider Architecture

Note that it is possible for a given name space, say DNS, to have more than one name space provider installed on a given machine.

As mentioned above, the generic term *service* refers to the server-half of a client/server application. In WinSock, a service is associated with a *service class*, and each instance of a particular service has a *service name* which must be unique within the service class. Examples of service classes include FTP Server, SQL Server, XYZ Corp. Employee Info Server, etc. As the example attempts to illustrate, some service classes are "well known" while others are very unique and specific to a particular vertical application. In either case, every service class is represented by both a class name and a class ID. The class name does not necessarily need to be unique, but the class ID must be. Globally Unique Identifiers (GUIDs) are used to represent service class IDs. For well-known services, class names and class ID's (GUIDs) have been pre-allocated, and macros are available to convert between, for example, TCP port numbers (in host byte order) and the corresponding class ID GUIDs. For other services, the developer chooses the class name and uses the **UUIDGEN.EXE** utility to generate a GUID for the class ID.

The notion of a service class exists to allow a set of attributes to be established that are held in common by all instances of a particular service. This set of attributes is provided at the time the service class is

defined to WinSock, and is referred to as the service class schema information. When a service is installed and made available on a host machine, that service is considered instantiated, and its service name is used to distinguish a particular instance of the service from other instances which may be known to the name space.

Note that the installation of a service class only needs to occur on machines where the service executes, not on all of the clients which may utilize the service. Where possible, the WinSock 2 DLL will provide service class schema information to a name space provider at the time an instantiation of a service is to be registered or a service query is initiated. The WinSock 2 DLL does not, of course, store this information itself, but attempts to retrieve it from a name space provider that has indicated its ability to supply this data. Since there is no guarantee that the WinSock 2 DLL will be able to supply the service class schema, name space providers that need this information must have a fallback mechanism to obtain it through name space-specific means.

As noted above, the Internet has adopted what can be termed a host-centric service model. Applications needing to locate the transport address of a service generally must first resolve the address of a specific host known to host the service. To this address they add in the well-known port number and thus create a complete transport address. To facilitate the resolution of host names, a special service class identifier has been pre-allocated (SVCID_HOSTNAME). A query that specifies SVCID_HOSTNAME as the service class and uses the host name the service instance name will, if the query is successful, return host address information.

In WinSock 2, applications that are protocol-independent wish to avoid the need to comprehend the internal details of a transport address. Thus the need to first get a host address and then add in the port is problematic. To avoid this, queries may also include the well-known name of a particular service and the protocol over which the service operates, such as “ftp/tcp”. In this case, a successful query will return a complete transport address for the specified service on the indicated host, and the application will not be required to “crack open” a sockaddr structure. This is described in more detail below.

The Internet’s Domain Name System does not have a well-defined means to store service class schema information. As a result, DNS name space providers will only be able to accommodate well-known TCP/IP services for which a service class GUID has been preallocated. In practice this is not a serious limitation since service class GUIDs have been preallocated for the entire set of TCP and UDP ports, and macros are available to retrieve the GUID associated with any TCP or UDP port (with the port expressed in host byte order). Thus all of the familiar services such as ftp, telnet, whois, etc. are well supported.

Continuing with our service class example, instance names of the ftp service may be “alder.intel.com” or “rhino.microsoft.com” while an instance of the XYZ Corp. Employee Info Server might be named “XYZ Corp. Employee Info Server Version 3.5”. In the first two cases, the combination of the service class GUID for ftp and the machine name (supplied as the service instance name) uniquely identify the desired service. In the third case, the host name where the service resides can be discovered at service query time, so the service instance name does not need to include a host name.

5.1.2. Summary of Name Resolution Functions

The name resolution functions can be grouped into three categories: Service installation, client queries, and helper functions (and macros). The sections that follow identify the functions in each category and briefly describe their intended use. Key data structures are also described.

5.1.2.1. Service Installation

- **WSAInstallServiceClass()**
- **WSARemoveServiceClass()**
- **WSASetService()**

When the required service class does not already exist, an application uses **WSAInstallServiceClass()** to install a new service class by supplying a service class name, a GUID for the service class ID, and a series of WSANSCLASSINFO structures. These structures are each specific to a particular name space, and supply common values such as recommended TCP port numbers or Netware SAP Identifiers. A service class can be removed by calling **WSARemoveServiceClass()** and supplying the GUID corresponding to the class ID.

Once a service class exists, specific instances of a service can be installed or removed via **WSASetService().** This function takes a WSAQUERYSET structure as an input parameter along with an operation code and operation flags. The operation code indicates whether the service is being installed or removed. The WSAQUERYSET structure provides all of the relevant information about the service including service class ID, service name (for this instance), applicable name space identifier and protocol information, and a set of transport addresses at which the service listens. Services should invoke **WSASetService()** when they initialize in order to advertise their presence in dynamic name spaces.

5.1.2.2. Client Query

- **WSAEnumNameSpaceProviders()**
- **WSALookupServiceBegin()**
- **WSALookupServiceNext()**
- **WSALookupServiceEnd()**

The **WSAEnumNameSpaceProviders()** function allows an application to discover which name spaces are accessible via WinSock's name resolution facilities. It also allows an application to determine whether a given name space is supported by more than one name space provider, and to discover the provider ID for any particular name space provider. Using a provider ID, the application can restrict a query operation to a specified name space provider.

WinSock's name space query operations involves a series of calls: **WSALookupServiceBegin()**, followed by one or more calls to **WSALookupServiceNext()** and ending with a call to **WSALookupServiceEnd()**. **WSALookupServiceBegin()** takes a WSAQUERYSET structure as input in order to define the query parameters along with a set of flags to provide additional control over the search operation. It returns a query handle which is used in the subsequent calls to **WSALookupServiceNext()** and **WSALookupServiceEnd()**.

The application invokes **WSALookupServiceNext()** to obtain query results, with results supplied in an application-supplied WSAQUERYSET buffer. The application continues to call **WSALookupServiceNext()** until the error code WSA_E_NO_MORE is returned indicating that all results have been retrieved. The search is then terminated by a call to **WSALookupServiceEnd()**. The **WSALookupServiceEnd()** function can also be used to cancel a currently pending **WSALookupServiceNext()** when called from another thread.

In WinSock 2, conflicting error codes are defined for WSAENOMORE (10102) and WSA_E_NO_MORE (10110). The error code WSAENOMORE will be removed in a future version and only WSA_E_NO_MORE will remain. For WinSock 2, however, applications should check for both WSAENOMORE and WSA_E_NO_MORE for the widest possible compatibility with Name Space Providers that use either one.

5.1.2.3. Helper Functions

- **WSAGetServiceClassNameByClassId()**
- **WSAAddressToString()**
- **WSAStringToAddress()**
- **WSAGetServiceClassInfo()**

The name resolution helper functions include a function to retrieve a service class name given a service class ID, a pair of functions used to translate a transport address between a **sockaddr** struct and an ASCII string representation, a function to retrieve the service class schema information for a given service class, and a set of macros for mapping well known services to pre-allocated GUIDs.

The following macros from winsock2.h aid in mapping between well known service classes and these name spaces.

SVCID_TCP(Port) SVCID_UDP(Port) SVCID_NETWARE(Object Type)	Given a port for TCP/IP or UDP/IP or the object type in the case of Netware, return the GUID (NOTE: the port number must be in host order)
IS_SVCID_TCP(GUID) IS_SVCID_UDP(GUID) IS_SVCID_NETWARE(GUID)	Returns TRUE if the GUID is within the allowable range
SET_TCP_SVCID(GUID, port) SET_UDP_SVCID(GUID, port)	Initializes a GUID structure with the GUID equivalent for a TCP or UDP port number (NOTE: the port number must be in host order)
PORT_FROM_SVCID_TCP(GUID) PORT_FROM_SVCID_UDP(GUID) SAPID_FROM_SVCID_NETWARE(GUID)	Returns the port or object type associated with the GUID (NOTE: the port number is in host order)

5.1.3. Name Resolution Data Structures

There are several important data structures that are used extensively throughout the name resolution functions. These are described below.

5.1.3.1. Query-Related Data Structures

The WSAQUERYSET structure is used to form queries for **WSALookupServiceBegin()**, and used to deliver query results for **WSALookupServiceNext()**. It is a complex structure since it contains pointers to several other structures, some of which reference still other structures. The relationship between WSAQUERYSET and the structures it references is illustrated as follows:

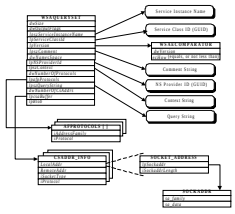

Figure 5 WSAQUERYSET and Friends

Within the WSAQUERYSET structure, most of the fields are self explanatory, but some deserve additional explanation. The *dwSize* field must always be filled in with sizeof(WSAQUERYSET), as this is used by name space providers to detect and adapt to different versions of the WSAQUERYSET structure that may appear over time.

The dwOutputFlags field is used by a name space provider to provide additional information about query results. For details, see **WSALookupServiceNext()**.

The WSAECOMPARATOR structure referenced by *lpversion* is used for both query constraint and results. For queries, the *dwVersion* field indicates the desired version of the service. The *ecHow* field is an enumerated type which specifies how the comparison will be made. The choices are

COMP_EQUALS which requires that an exact match in version occurs, or COMP_NOTLESS which specifies that the service's version number be no less than the value of *dwVersion*.

The interpretation of *dwNameSpace* and *lpNSProviderId* depends upon how the structure is being used and is described further in the individual function descriptions that utilize this structure.

The *lpszContext* field applies to hierarchical name spaces, and specifies the starting point of a query or the location within the hierarchy where the service resides. The general rules are:

- A value of NULL, blank ("") will start the search at the default context.
- A value of "\" starts the search at the top of the name space.
- Any other value starts the search at the designated point.

Providers that do not support containment may return an error if anything other than "" or "\" is specified. Providers that support limited containment, such as groups, should accept "", '\', or a designated point. Contexts are name space specific. If *dwNameSpace* is NS_ALL, then only "" or "\" should be passed as the context since these are recognized by all name spaces.

The *lpszQueryString* field is used to supply additional, name space-specific query information such as a string describing a well-known service and transport protocol name, as in "ftp/tcp".

The AFPROTOCOLS structure referenced by *lpafpProtocols* is used for query purposes only, and supplies a list of protocols to constrain the query. These protocols are represented as (address family, protocol) pairs, since protocol values only have meaning within the context of an address family.

The array of CSADDR_INFO structure referenced by *lpcsaBuffer* contain all of the information needed to for either a service to use in establishing a listen, or a client to use in establishing a connection to the service. The *LocalAddr* and *RemoteAddr* fields both directly contain a SOCKET_ADDRESS structure. A service would create a socket using the tuple (*LocalAddr.lpSockaddr->sa_family, iSocketType, iProtocol*). It would bind the socket to a local address using *LocalAddr.lpSockaddr*, and *LocalAddr.lpSockaddrLength*. The client creates its socket with the tuple (*RemoteAddr.lpSockaddr->sa_family, iSocketType, iProtocol*), and uses the combination of *RemoteAddr.lpSockaddr*, and *RemoteAddr.lpSockaddrLength* when making a remote connection.

5.1.3.2. Service Class Data Structures

When a new service class is installed, a WSASERVICECLASSINFO structure must be prepared and supplied. This structure also consists of substructures which contain a series of parameters that apply to specific name spaces.

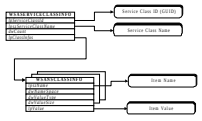

Figure 6 Class Info Data Structures

For each service class, there is a single WSASERVICECLASSINFO structure. Within the WSASERVICECLASSINFO structure, the service class' unique identifier is contained in *lpServiceClassId*, and an associated display string is referenced by *lpServiceClassName*. This is the string that will be returned by **WSAGetServiceClassNameByServiceClassId()**.

The *lpClassInfos* field in the WSASERVICECLASSINFO structure references an array of WSANSCLASSINFO structures, each of which supplies a named and typed parameter that applies to a specified name space. Examples of values for the *lpszName* field include: "SapId", "TcpPort", "UdpPort", etc. These strings are generally specific to the name space identified in *dwNameSpace*.

Typical values for *dwValueType* might be REG_DWORD, REG_SZ, etc. The *dwValueSize* field indicates the length of the data item pointed to by *lpValue*.

The entire collection of data represented in a WSASERVICECLASSINFO structure is provided to each name space provider when **WSAInstallServiceClass()** is invoked. Each individual name space provider then sifts through the list of WSANSCLASSINFO structures and retain the information applicable to it.

5.2. Name Resolution Function Reference

5.2.1. WSAAddressToString()

Description **WSAAddressToString()** converts all components of a SOCKADDR structure into a human-readable numeric string representation of the address. This is intended to be used mainly for display purposes. If the caller wishes the translation to be done by a particular provider, it should supply the corresponding WSAPROTOCOL_INFO struct in the *lpProtocolInfo* parameter.

```
INT WSAAPI
WSAAddressToString(
        IN          LPSOCKADDR              lpsaAddress,
        IN          DWORD                   dwAddressLength,
        IN          LPWSAPROTOCOL_INFO      lpProtocolInfo,
        OUT         LPTSTR                  lpszAddressString,
        IN OUT      LPDWORD                 lpdwAddressStringLength
);
```

lpsaAddress points to a **SOCKADDR** structure to translate into a string.

dwAddressLength the length of the Address **SOCKADDR** (which may vary in size with different protocols)

lpProtocolInfo (optional) a WSAPROTOCOL_INFO struct associated with the provider to be used. If this is NULL, the call is routed to the provider of the first protocol supporting the address family indicated in *lpsaAddress*.

lpszAddressString a buffer which receives the human-readable address string.

lpdwAddressStringLength on input, the length of the AddressString buffer. On output, returns the length of the string actually copied into the buffer. If the supplied buffer is not large enough, the function fails with a specific error of WSAEFAULT and this parameter is updated with the required size in bytes.

Return Value The return value is 0 if the operation was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Errors

WSAEFAULT	The specified *lpcsAddress, lpProtocolInfo, lpszAddressString* are not all in the process' address space, or the *lpszAddressString* buffer is too small. Pass in a larger buffer
WSAEINVAL	The specified Address is not a valid socket address, or there was no transport provider supporting its indicated address family.

WSANOTINITIALIZED	The Winsock 2 DLL has not been initialized. The application must first call **WSAStartup()** before calling any WinSock functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

5.2.2. WSAEnumNameSpaceProviders()

Description Retrieve information about available name spaces.

```
INT WSAAPI
WSAEnumNameSpaceProviders (
    IN OUT LPDWORD                  lpdwBufferLength,
    OUT    LPWSANAMESPACE_INFO      lpnspBuffer
);
```

lpdwBufferLength on input, the number of bytes contained in the buffer pointed to by *lpnspBuffer*. On output (if the API fails, and the error is WSAEFAULT), the minimum number of bytes to pass for the *lpnspBuffer* to retrieve all the requested information. The passed-in buffer must be sufficient to hold all of the name space information.

lpnspBuffer A buffer which is filled with **WSANAMESPACE_INFO** structures described below. The returned structures are located consecutively at the head of the buffer. Variable sized information referenced by pointers in the structures point to locations within the buffer located between the end of the fixed sized structures and the end of the buffer. The number of structures filled in is the return value of **WSAEnumNameSpaceProviders().**

Data Types The following data types are used in this call.
The **WSANAMESPACE_INFO** structure contains all of the registration information for a name space provider.

```
typedef struct _WSANAMESPACE_INFO {
GUID            NSProviderId;
DWORD           dwNameSpace;
BOOL            fActive;
DWORD           dwVersion;
LPTSTR          lpszIdentifier;
} WSANAMESPACE_INFO, *PWSANAMESPACE_INFO,
    *LPWSANAMESPACE_INFO;
```

NSProviderId The unique identifier for this name space provider.

dwNameSpace Specifies the name space supported by this implementation of the provider.

fActive If TRUE, indicates that this provider is active. If FALSE, the provider is inactive and is not accessible for queries, even if the query specifically references this provider.

dwVersion Name Space version identifier.

lpszIdentifier Display string for the provider.

Return Value **WSAEnumNameSpaceProviders()** returns the number of **WSANAMESPACE_INFO** structures copied into *lpnspBuffer*. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Errors

WSAEFAULT	the buffer length was too small to receive all the relevant **WSANAMESPACE_INFO** structures and associated information. Pass in a buffer at least as large as the value returned in *lpdwBufferLength*.
WSANOTINITIALIZED	The Winsock 2 DLL has not been initialized. The application must first call **WSAStartup()** before calling any WinSock functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

5.2.3. WSAGetServiceClassInfo

Description **WSAGetServiceClassInfo()** is used to retrieve all of the class information (schema) pertaining to a specified service class from a specified name space provider.

```
INT WSAAPI
WSAGetServiceClassInfo(
    IN      LPGUID                  lpProviderId,
    IN      LPGUID                  lpServiceClassId,
    IN OUT  LPDWORD                 lpdwBufferLength,
    OUT     LPWSASERVICECLASSINFO   lpServiceClassInfo
);
```

lpProviderId Pointer to a GUID which identifies a specific name space provider

lpServiceClassId Pointer to a GUID identifying the service class in question

lpdwBufferLength on input, the number of bytes contained in the buffer pointed to by *lpServiceClassInfos*. On output - if the API fails, and the error is WSAEFAULT, then it contains the minimum number of bytes to pass for the *lpServiceClassInfo* to retrieve the record.

lpServiceClasslnfo returns service class information from the indicated name space provider for the specified service class.

Remarks The service class information retrieved from a particular name space provider may not necessarily be the complete set of class information that was supplied when the service class was installed. Individual name space providers are only required to retain service class information that is applicable to the name spaces that they support. See section *5.1.3.2. Service Class Data Structures*for more information.

Return Value The return value is 0 if the operation was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Errors

WSAEACCES	The calling routine does not have sufficient privileges to access the information.
WSAEFAULT	The buffer referenced by *lpServiceClassInfo* is too small. Pass in a larger buffer.
WSAEINVAL	the specified service class ID or name space provider ID is invalid.
WSANOTINITIALIZED	The Winsock 2 DLL has not been initialized. The application must first call **WSAStartup()** before calling any WinSock functions.
WSATYPE_NOT_FOUND	The specified class was not found.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

5.2.4. WSAGetServiceClassNameByClassId()

Description This API will return the name of the service associated with the given type. This name is the generic service name, like FTP, or SNA, and not the name of a specific instance of that service.

```
INT WSAAPI
WSAGetServiceClassNameByClassId(
        IN          LPGUID          lpServiceClassId,
        OUT         LPTSTR          lpszServiceClassName,
        IN OUT      LPDWORD         lpdwBufferLength
);
```

lpServiceClassId pointer to the GUID for the service class.

lpszServiceClassName service name.

lpdwBufferLength on input length of buffer returned by *lpszServiceClassName*. On output, the length of the service name copied into *lpszServiceClassName*.

Return Value The return value is 0 if the operation was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Errors

WSAEFAULT	The specified ServiceClassName buffer is too small. Pass in a larger buffer
WSAEINVAL	the specified ServiceClassId is invalid.
WSANOTINITIALIZED	The Winsock 2 DLL has not been initialized. The application must first call **WSAStartup()** before calling any WinSock functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

5.2.5. WSAInstallServiceClass()

Description **WSAInstallServiceClass()** is used to register a service class schema within a name space. This schema includes the class name, class id, and any name space specific information that is common to all instances of the service, such as the SAP ID or object ID.

```
INT WSAAPI
WSAInstallServiceClass(
        IN      LPWSASERVICECLASSINFO   lpServiceClassInfo,
);
```

lpServiceClasslnfo contains service class to name space specific type mapping information. Multiple mappings can be handled at one time.

See section *5.1.3.2. Service Class Data Structures* for a description of pertinent data structures.

Return Value The return value is 0 if the operation was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Errors

WSAEACCES	The calling routine does not have sufficient privileges to install the Service.
WSAEALREADY	Service class information has already been registered for this service class ID. To modify service class info, first use **WSARemoveServiceClass()**, and then re-install with updated class info data.
WSAEINVAL	The service class information was invalid or improperly structured.
WSANOTINITIALIZED	The Winsock 2 DLL has not been initialized. The application must first call **WSAStartup()** before calling any WinSock functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

5.2.6. WSALookupServiceBegin()

Description **WSALookupServiceBegin()** is used to initiate a client query that is constrained by the information contained within a WSAQUERYSET structure. **WSALookupServiceBegin()** only returns a handle, which should be used by subsequent calls to **WSALookupServiceNext()** to get the actual results.

```
INT WSAAPI
WSALookupServiceBegin (
        IN      LPWSAQUERYSET           lpqsRestrictions,
        IN      DWORD                   dwControlFlags,
        OUT     LPHANDLE                lphLookup
);
```

lpqsRestrictions contains the search criteria. See below for details.

dwControlFlags controls the depth of the search.

LUP_DEEP	Query deep as opposed to just the first level.
LUP_CONTAINERS	Return containers only
LUP_NOCONTAINERS	Don't return any containers
LUP_FLUSHCACHE	If the provider has been caching information, ignore the cache and go query the name space itself.
LUP_FLUSHPREVIOUS	Used as a value for the *dwControlFlags* argument in **WSALookupServiceNext()**. Setting this flag instructs the provider to discard the last result set, which was too large for the supplied buffer, and move on to the next result set.
LUP_NEAREST	If possible, return results in the order of distance. The measure of distance is provider specific.
LUP_RES_SERVICE	indicates whether prime response is in the remote or local part of CSADDR_INFO structure. The other part needs to be "useable" in either case.
LUP_RETURN_ALIASES	Any available alias information is to be returned in successive calls to **WSALookupServiceNext()**, and each alias returned will have the RESULT_IS_ALIAS flag set.
LUP_RETURN_NAME	Retrieve the name as *lpszServiceInstanceName*
LUP_RETURN_TYPE	Retrieve the type as *lpServiceClassId*
LUP_RETURN_VERSION	Retrieve the version as *lpVersion*
LUP_RETURN_COMMENT	Retrieve the comment as *lpszComment*
LUP_RETURN_ADDR	Retrieve the addresses as *lpcsaBuffer*
LUP_RETURN_BLOB	Retrieve the private data as *lpBlob*
LUP_RETURN_QUERY_STRING	Retrieve unparsed remainder of the service instance name as *lpszQueryString*
LUP_RETURN_ALL	Retrieve all of the information

lphLookup Handle to be used when calling **WSALookupServiceNext** in order to start retrieving the results set.

Remarks If LUP_CONTAINERS is specified in a call, all other restriction values should be avoided. If any are supplied, it is up to the name service provider to decide if it can support this restriction over the containers. If it cannot, it should return an error.

Some name service providers may have other means of finding containers. For example, containers might all be of some well-known type, or of a set of well-known types, and therefore a query restriction may be created for finding them. No matter what other means the name service provider has for locating containers, LUP_CONTAINERS and LUP_NOCONTAINERS take precedence. Hence, if a query restriction is given that includes containers, specifying LUP_NOCONTAINERS will prevent the container items from being returned. Similarly, no matter the query restriction, if LUP_CONTAINERS is given, only containers should be returned. If a name space does not support containers, and LUP_CONTAINERS is specified, it should simply return WSANO_DATA.

The preferred method of obtaining the containers within another container, is the call:

```
dwStatus = WSALookupServiceBegin(
                  lpqsRestrictions,
                  LUP_CONTAINERS,
                  lphLookup);
```

followed by the requisite number of **WSALookupServiceNext** calls. This will return all containers contained immediately within the starting context; that is, it is not a deep query. With this, one can map the address space structure by walking the hierarchy, perhaps enumerating the content of selected containers. Subsequent uses of **WSALookupServiceBegin** use the containers returned from a previous call.

Forming Queries

As mentioned above, a WSAQUERYSET structure is used as an input parameter to **WSALookupBegin()** in order to qualify the query. The following table indicates how the WSAQUERYSET is used to construct a query. When a field is marked as *(Optional)* a NULL pointer may be supplied, indicating that the field will not be used as a search criteria. See section *5.1.3.1. Query-Related Data Structures* for additional information.

WSAQUERYSET Field Name	Query Interpretation
dwSize	Must be set to sizeof(WSAQUERYSET). This is a versioning mechanism.
DwOuputFlags	Ignored for queries
lpszServiceInstanceName	*(Optional)* Referenced string contains service name. The semantics for wildcarding within the string are not defined, but may be supported by certain name space providers.
LpServiceClassId	(*Required)* The GUID corresponding to the service class.
lpVersion	*(Optional)* References desired version number and provides version comparison semantics (i.e. version must match exactly, or version must be not less than the value supplied).

LpszComment	Ignored for queries.
DwNameSpace	Identifier of a single name space in which to constrain the search, or NS_ALL to include all name spaces. See important note below.
LpNSProviderId	*(Optional)* References the GUID of a specific name space provider, and limits the query to this provider only.
LpszContext	*(Optional)* Specifies the starting point of the query in a hierarchical name space.
DwNumberOfProtocols	Size of the protocol constraint array, may be zero.
LpafpProtocols	*(Optional)* References an array of AFPROTOCOLS structure. Only services that utilize these protocols will be returned.
LpszQueryString	*(Optional)* Some namespaces (such as whois++) support enriched SQL like queries which are contained in a simple text string. This parameter is used to specify that string.
DwNumberOfCsAddrs	Ignored for queries.
LpcsaBuffer	Ignored for queries.
LpBlob	*(Optional)* This is a pointer to a provider-specific entity.

Important Note:

In most instances, applications interested in only a particular transport protocol should constrain their query by address family and protocol rather than by name space. This would allow an application that wishes to locate a TCP/IP service, for example, to have its query processed by all available name spaces such as the local hosts file, DNS, NIS, etc.

Return Value The return value is 0 if the operation was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Errors

WSAEINVAL	One or more parameters were invalid for this provider or missing.
WSANO_DATA	The name was found in the database but no data matching the given restrictions was located..
WSANOTINITIALIZED	The Winsock 2 DLL has not been initialized. The application must first call **WSAStartup()** before calling any WinSock functions.
WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

5.2.7. WSALookupServiceEnd()

Description **WSALookupServiceEnd()** is called to free the handle after previous calls to **WSALookupServiceBegin()** and **WSALookupServiceNext()**. Note that if you call **WSALookupServiceEnd()** from another thread while an existing **WSALookupServiceNext()** is blocked, then the end call will have the same effect as a cancel, and will cause the **WSALookupServiceNext()** call to return immediately.

```
INT WSAAPI
WSALookupServiceEnd (
        IN      HANDLE      hLookup,
);
```

hLookup Handle previously obtained by calling **WSALookupServiceBegin()**.

Return Value The return value is 0 if the operation was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Errors

WSA_INVALID_HANDLE	The Handle is not valid
WSANOTINITIALIZED	The Winsock 2 DLL has not been initialized. The application must first call **WSAStartup()** before calling any WinSock functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

5.2.8. WSALookupServiceNext()

Description **WSALookupServiceNext()** is called after obtaining a Handle from a previous call to **WSALookupServiceBegin()** in order to retrieve the requested service information. The provider will pass back a **WSAQUERYSET** structure in the *lpqsResults* buffer. The client should continue to call this API until it returns WSA_E_NOMORE, indicating that all of the **WSAQUERYSET** have been returned.

```
INT WSAAPI
WSALookupServiceNext (
        IN          HANDLE          hLookup,
        IN          DWORD           dwControlFlags,
        IN OUT      LPDWORD         lpdwBufferLength,
        OUT         LPWSAQUERYSET   lpqsResults
);
```

hLookup Handle returned from the previous call to **WSALookupServiceBegin()**.

dwControlFlags Flags to control the next operation. Currently only LUP_FLUSHPREVIOUS is defined as a means to cope with a result set which is too large. If an application does not wish to (or cannot) supply a large enough buffer, setting LUP_FLUSHPREVIOUS instructs the provider to discard the last result set - which was too large - and move on to the next set for this call.

lpdwBufferLength on input, the number of bytes contained in the buffer pointed to by *lpqsResults*. On output - if the API fails, and the error is WSAEFAULT, then it contains the minimum number of bytes to pass for the *lpqsResults* to retrieve the record.

lpqsResults a pointer to a block of memory, which will contain one result set in a **WSAQUERYSET** structure on return.

Remarks The *dwControlFlags* specified in this function and the ones specified at the time of **WSALookupServiceBegin()** are treated as "restrictions" for the purpose of combination. The restrictions are combined between the ones at **WSALookupServiceBegin()** time and the ones at **WSALookupServiceNext()** time. Therefore the flags at **WSALookupServiceNext()** can never increase the amount of data returned beyond what was requested at **WSALookupServiceBegin()**, although it is NOT an error to specify more or fewer flags. The flags specified at a given **WSALookupServiceNext()** apply only to that call.

The *dwControlFlags* LUP_FLUSHPREVIOUS and LUP_RES_SERVICE are exceptions to the "combined restrictions" rule (because they are "behavior" flags instead of "restriction" flags). If either of these flags are used in **WSALookupServiceNext()** they have their defined effect regardless of the setting of the same flags at **WSALookupServiceBegin()**.

For example, if LUP_RETURN_VERSION is specified at **WSALookupServiceBegin()** the service provider retrieves records including the "version". If LUP_RETURN_VERSION is NOT specified at **WSALookupServiceNext()**, the

returned information does not include the “version”, even though it was available. No error is generated.

Also for example, if LUP_RETURN_BLOB is NOT specified at **WSALookupServiceBegin()** but is specified at **WSALookupServiceNext()**, the returned information does not include the private data. No error is generated.

Query Results

The following table describes how the query results are represented in the WSAQUERYSET structure. Refer to section *5.1.3.1. Query-Related Data Structures* for additional information.

WSAQUERYSET Field Name	Result Interpretation
dwSize	Will be set to sizeof(WSAQUERYSET). This is used as a versioning mechanism.
DwOuputFlags	RESULT_IS_ALIAS flag indicates this is an alias result.
lpszServiceInstanceName	Referenced string contains service name.
LpServiceClassId	The GUID corresponding to the service class.
lpVersion	References version number of the particular service instance.
LpszComment	Optional comment string supplied by service instance.
DwNameSpace	Name space in which the service instance was found.
LpNSProviderId	Identifies the specific name space provider that supplied this query result.
lpszContext	Specifies the context point in a hierarchical name space at which the service is located.
DwNumberOfProtocols	Undefined for results.
LpafpProtocols	Undefined for results, all needed protocol information is in the CSADDR_INFO structures.
LpszQueryString	When *dwControlFlags* includes LUP_RETURN_QUERY_STRING, this field returns the unparsed remainder of the *lpszServiceInstanceName* specified in the original query. For example, in a name space that identifies services by hierarchical names that specify a host name and a file path within that host, the address returned might be the host address and the unparsed remainder might be the file path. If the *lpszServiceInstanceName* is fully parsed and LUP_RETURN_QUERY_STRING is used, this field is NULL or points to a zero-length string.
DwNumberOfCsAddrs	Indicates the number of elements in the array of CSADDR_INFO structures.
LpcsaBuffer	A pointer to an array of CSADDR_INFO structures, with one complete transport address contained within each element.

LpBlob	*(Optional)* This is a pointer to a provider-specific entity.

Return Value The return value is 0 if the operation was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Errors

WSA_E_NO_MORE	There is no more data available. In WinSock 2, conflicting error codes are defined for WSAENOMORE (10102) and WSA_E_NO_MORE (10110). The error code WSAENOMORE will be removed in a future version and only WSA_E_NO_MORE will remain. For WinSock 2, however, applications should check for both WSAENOMORE and WSA_E_NO_MORE for the widest possible compatibility with Name Space Providers that use either one.
WSA_E_CANCELLED	A call to **WSALookupServiceEnd()** was made while this call was still processing. The call has been canceled. The data in the *lpqsResults* buffer is undefined. In WinSock 2, conflicting error codes are defined for WSAECANCELLED (10103) and WSA_E_CANCELLED (10111). The error code WSAECANCELLED will be removed in a future version and only WSA_E_CANCELLED will remain. For WinSock 2, however, applications should check for both WSAECANCELLED and WSA_E_CANCELLED for the widest possible compatibility with Name Space Providers that use either one.
WSAEFAULT	The *lpqsResults* buffer was too small to contain a **WSAQUERYSET** set.
WSAEINVAL	One or more required parameters were invalid or missing.
WSA_INVALID_HANDLE	The specified Lookup handle is invalid.
WSANOTINITIALIZED	The Winsock 2 DLL has not been initialized. The application must first call **WSAStartup()** before calling any WinSock functions.
WSANO_DATA	The name was found in the database but no data matching the given restrictions was located..
WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space.

WSA_NOT_ENOUGH_MEMORY

There was insufficient memory to perform the operation.

5.2.9. WSARemoveServiceClass()

Description **WSARemoveServiceClass()** is used to permanently unregister service class schema.

```
INT WSAAPI
WSARemoveServiceClass(
        IN      LPGUID      lpServiceClassId
);
```

lpServiceClassId Pointer to the GUID for the service class that you wish to remove.

Return Value The return value is 0 if the operation was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Errors

WSATYPE_NOT_FOUND	The specified class was not found.
WSAEACCES	The calling routine does not have sufficient privileges to remove the Service.
WSANOTINITIALIZED	The Winsock 2 DLL has not been initialized. The application must first call **WSAStartup()** before calling any WinSock functions.
WSAEINVAL	The specified GUID was not valid.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

4.2.10. WSASetService()

Description **WSASetService()** is used to register or deregister a service instance within one or more name spaces. This function may be used to affect a specific name space provider, all providers associated with a specific name space, or all providers across all name spaces.

```
INT WSAAPI
WSASetService(
        IN      LPWSAQUERYSET           lpqsRegInfo,
        IN      WSAESETSERVICEOP        essOperation,
        IN      DWORD                   dwControlFlags
);
```

lpqsRegInfo specifies service information for registration, identifies service for deregistration.

essOperation an enumeration whose values include:

RNRSERVICE_REGISTER register the service. For SAP, this means sending out a periodic broadcast. This is a NOP for the DNS name space. For persistent data stores this means updating the address information.

RNRSERVICE_DEREGISTER deregister the service. For SAP, this means stop sending out the periodic broadcast. This is a NOP for the DNS name space. For persistent data stores this means deleting address information.

RNRSERVICE_DELETE delete the service from dynamic name and persistent spaces. For services represented by multiple CSADDR_INFO structures (using the SERVICE_MULTIPLE flag), only the supplied address will be deleted, and this much match exactly the corresponding CSADD_INFO structure that was supplied when the service was registered.

dwControlFlags A set of flags whose values include:

SERVICE_MULTIPLE Controls scope of operation. When clear, service addresses are managed as a group. A register or deregister invalidates all existing addresses before adding the given address set. When set, the action is only performed on the given address set. A register does not invalidate existing addresses and a deregister only invalidates the given set of addresses.

The available values for *essOperation* and *dwControlFlags* combine to give meanings as shown in the following table:

Operation	Flags	Service already exists	Service does not exist
RNRSERVICE_ REGISTER	none	Overwrite the object. Use only addresses specified. Object is REGISTERED.	Create a new object. Use only addresses specified. Object is REGISTERED.
RNRSERVICE_ REGISTER	SERVICE_ MULTIPLE	Update object. Add new addresses to existing set.	Create a new object. Use all addresses

		Object is REGISTERED.	specified. Object is REGISTERED.
RNRSERVICE_DEREGISTER	none	Remove all addresses, but do not remove object from name space. Object is DEREGISTERED.	WSASERVICE_NOT_FOUND
RNRSERVICE_DEREGISTER	SERVICE_MULTIPLE	Update object. Remove only addresses that are specified. Only mark object as DEREGISTERED if no addresses present. Do not remove from the name space.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	none	Remove object from the name space.	WSASERVICE_NOT_FOUND
RNRSERVICE_DELETE	SERVICE_MULTIPLE	Remove only addresses that are specified. Only remove object from the name space if no addresses remain.	WSASERVICE_NOT_FOUND

Remarks

SERVICE_MULTIPLE lets an application manage its addresses independently. This is useful when the application wants to manage its protocols individually or when the service resides on more than one machine. For instance, when a service uses more than one protocol, it may find that one listening socket aborts but the others remain operational. In this case, the service could deregister the aborted address without affecting the other addresses.

When using SERVICE_MULTIPLE, an application must not let stale addresses remain in the object. This can happen if the application aborts without issuing a DEREGISTER request. When a service registers, it should store its addresses. On its next invocation, the service should explicitly deregister these old stale addresses before registering new addresses.

Service Properties

The following table describes how service property data is represented in a WSAQUERYSET structure. Fields labeled as *(Optional)* may be supplied with a NULL pointer.

WSAQUERYSET Field Name	Service Property Description
dwSize	Must be set to sizeof(WSAQUERYSET). This is a versioning mechanism.
DwOuputFlags	Not applicable and ignored.
LpszServiceInstanceName	Referenced string contains the service instance name.
LpServiceClassId	The GUID corresponding to this service class.
lpVersion	*(Optional)* Supplies service instance version number.
LpszComment	*(Optional)* An optional comment string.

DwNameSpace	See table below.
LpNSProviderId	See table below.
LpszContext	*(Optional)* Specifies the starting point of the query in a hierarchical name space.
DwNumberOfProtocols	Ignored.
LpafpProtocols	Ignored.
LpszQueryString	Ignored.
DwNumberOfCsAddrs	The number of elements in the array of CSADDRO_INFO structs referenced by *lpcsaBuffer*.
LpcsaBuffer	A pointer to an array of CSADDRO_INFO structs which contain the address[es] that the service is listening on.
lpBlob	*(Optional)* This is a pointer to a provider-specific entity.

As illustrated below, the combination of the *dwNameSpace* and *lpNSProviderId* parameters determine which name space providers are affected by this function.

DwNameSpace	***lpNSProviderId***	**Scope of Impact**
Ignored	Non-NULL	The specified name space provider
a valid name space ID	NULL	All name space providers that support the indicated name space
NS_ALL	NULL	All name space providers

Return Value The return value is 0 if the operation was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Errors

WSAEACCES	The calling routine does not have sufficient privileges to install the Service.
WSAEINVAL	One or more required parameters were invalid or missing.
WSANOTINITIALIZED	The Winsock 2 DLL has not been initialized. The application must first call **WSAStartup()** before calling any WinSock functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.
WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space.

5.2.10. WSAStringToAddress()

Description **WSAStringToAddress()** converts a human-readable numeric string to a socket address structure (SOCKADDR) suitable for passing to Windows Sockets routines which take such a structure. Any missing components of the address will be defaulted to a reasonable value if possible. For example, a missing port number will be defaulted to zero. If the caller wishes the translation to be done by a particular provider, it should supply the corresponding WSAPROTOCOL_INFO struct in the *lpProtocolInfo* parameter.

```
INT WSAAPI
WSAStringToAddress(
        IN          LPTSTR                  AddressString,
        IN          INT                     AddressFamily,
        IN          LPWSAPROTOCOL_INFO      lpProtocolInfo,
        OUT         LPSOCKADDR              lpAddress,
        IN OUT      LPINT                   lpAddressLength
);
```

AddressString points to the zero-terminated human-readable string to convert.

AddressFamily the address family to which the string belongs.

lpProtocolInfo (optional) the WSAPROTOCOL_INFO struct associated with the provider to be used. If this is NULL, the call is routed to the provider of the first protocol supporting the indicated *AddressFamily*.

Address a buffer which is filled with a single **SOCKADDR** structure.

lpAddressLength The length of the Address buffer. Returns the size of the resultant **SOCKADDR** structure. If the supplied buffer is not large enough, the function fails with a specific error of WSAEFAULT and this parameter is updated with the required size in bytes.

Return Value The return value is 0 if the operation was successful. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Errors

WSAEFAULT	The specified Address buffer is too small. Pass in a larger buffer.
WSAEINVAL	Unable to translate the string into a **SOCKADDR** or there was no transport provider supporting the indicated address family.
WSANOTINITIALIZED	The Winsock 2 DLL has not been initialized. The application must first call **WSAStartup()** before calling any WinSock functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

5.3. WinSock 1.1 Compatibile Name Resolution for TCP/IP

5.3.1. Introduction

Windows Sockets 1.1 defined a number of routines that were used for name resolution with TCP/IP (IP version 4) networks. These are customarily referred to as the **getXbyY()** functions and include the following:

- **gethostname()**
- **gethostbyaddr()**
- **gethostbyname()**
- **getprotobyname()**
- **getprotobynumber()**
- **getservbyname()**
- **getservbyport()**

Asynchronous versions of these functions were also defined:

- **WSAAsyncGetHostByAddr()**
- **WSAAsyncGetHostByName()**
- **WSAAsyncGetProtoByName()**
- **WSAAsyncGetProtoByNumber()**
- **WSAAsyncGetServByName()**
- **WSAAsyncGetSetvByPort()**

There are also two functions (now implemented in the WinSock 2 DLL) used to convert dotted IPv4 internet address notation to and from string and binary representations, respectively:

- **inet_addr()**
- **inet_ntoa()**

All of these functions are specific to IPv4 TCP/IP networks and developers of protocol-independent applications are discouraged from continuing to utilize these transport-specific functions. However, in order to retain strict backwards compatibility with WinSock 1.1, all of the above functions continue to be supported as long as at least one name space provider is present that supports the AF_INET address family (these functions are not relevant to IP version 6, denoted by AF_INET6).

The WinSock 2 DLL implements these compatibility functions in terms of the new, protocol-independent name resolution facilities using an appropriate sequence of **WSALookupServiceBegin/Next/End()** function calls. The details of how the **getXbyY()** functions are mapped to name resolution functions are provided below. Note that the WinSock 2 DLL handles the differences between the asynchronous and synchronous versions of the **getXbyY()** functions, so only the implementation of the synchronous **getXbyY()** functions are discussed.

5.3.2. Basic Approach

Most **getXbyY()** functions are translated by the WinSock 2 DLL to a **WSAServiceLookupBegin/Next/End()** sequence that uses one of a set of special GUIDs as the service class. These GUIDs identify the type of getXbyYoperation that is being emulated. The query is constrained to those NSPs that support AF_INET. Whenever a getXbyY function returns a hostent or servent structure, the WinSock 2 DLL will specify the LUP_RETURN_BLOB flag in **WSALookupServiceBegin()** so that the desired information will be returned by the NSP. These structures must be modified slightly in that the pointers contained within must be replaced with offsets that are relative to the start of the blob's data. All values referenced by these pointer fields must, of course, be completely contained within the blob, and all strings are ASCII.

5.3.3. getprotobyname and getprotobynumber

These functions are implemented within the WinSock 2 DLL by consulting a local protocols database. They do not result in any name resolution query.

5.3.4. getservbyname() and getservbyport()

The **WSALookupServiceBegin()** query uses SVCID_INET_SERVICEBYNAME as the service class GUID. The *lpszServiceInstanceName* field references a string which indicates the service name or service port, and (optionally) the service protocol. The formatting of the string is illustrated as "ftp/tcp" or "21/tcp" or just "ftp". The string is not case sensitive. The slash mark, if present, separates the protocol identifier from the preceding part of the string. The WinSock 2 DLL will specify LUP_RETURN_BLOB and the NSP will place a servent struct in the blob (using offsets instead of pointers as described above). NSPs should honor these other LUP_RETURN_* flags as well:

LUP_RETURN_NAME -> return the s_name field from servent struct in *lpszServiceInstanceName*

LUP_RETURN_TYPE -> return canonical GUID in lpServiceClassId It is understood that a service identified either as "ftp" or "21" may in fact be on some other port according to locally established conventions. The *s_port* field of the servent struct should indicate where the service can be contacted in the local environment. The canonical GUID returned when LUP_RETURN_TYPE is set should be one of the predefined GUID from svcs.h that corresponds to the port number indicated in the servent structure.

5.3.5. gethostbyname()

The **WSALookupServiceBegin()** query uses SVCID_INET_HOSTADDRBYNAME as the service class GUID. The host name is supplied in *lpszServiceInstanceName*. The WinSock 2 DLL specifies LUP_RETURN_BLOB and the NSP places a hostent struct in the blob (using offsets instead of pointers as described above). NSPs should honor these other LUP_RETURN_* flags as well:

LUP_RETURN_NAME -> return the *h_name* field from hostent struct in *lpszServiceInstanceName*

LUP_RETURN_ADDR -> return addressing info from hostent in CSADDR_INFO structs, port information is defaulted to zero. Note that this routine does **not** resolve host names that consist of a dotted internet address.

5.3.6. gethostbyaddr()

The **WSALookupServiceBegin()** query uses SVCID_INET_HOSTNAMEBYADDR as the service class GUID. The host address is supplied in *lpszServiceInstanceName* as a dotted internet string (e.g. "192.9.200.120"). The WinSock 2 DLL specifies LUP_RETURN_BLOB and the NSP places a hostent struct in the blob (using offsets instead of pointers as described above). NSPs should honor these other LUP_RETURN_* flags as well:

LUP_RETURN_NAME -> return the h_name field from hostent struct in *lpszServiceInstanceName*

LUP_RETURN_ADDR -> return addressing info from hostent in CSADDR_INFO structs, port information is defaulted to zero

5.3.7. gethostname()

The **WSALookupServiceBegin()** query uses SVCID_HOSTNAME as the service class GUID. If *lpszServiceInstanceName* is NULL or references a NULL string (i.e. ""), the local host is to be resolved. Otherwise, a lookup on a specified host name shall occur. For the purposes of emulating **gethostname()** the WinSock 2 DLL will specify a null pointer for *lpszServiceInstanceName*, and specify LUP_RETURN_NAME so that the host name is returned in the *lpszServiceInstanceName* field. If an application uses this query and specifies LUP_RETURN_ADDR then the host address will be provided in

a CSADDR_INFO struct. The LUP_RETURN_BLOB action is undefined for this query. Port information will be defaulted to zero unless the *lpszQueryString* references a service such as "ftp", in which case the complete transport address of the indicated service will be supplied.

5.4. WinSock 1.1 Compatible Name Resolution Reference

5.4.1. gethostbyaddr()

Description Get host information corresponding to an address.

```
#include <winsock2.h>

struct hostent FAR * WSAAPI
gethostbyaddr (
        IN      const char FAR *        addr,
        IN      int                     len,
        IN      int                     type
);
```

addr A pointer to an address in network byte order.

len The length of the address.

type The type of the address.

Remarks **gethostbyaddr()** returns a pointer to the following structure which contains the name(s) and address which correspond to the given address. All strings are null terminated.

```
struct hostent {
      char FAR *  h_name;
      char FAR * FAR *  h_aliases;
      short h_addrtype;
      short h_length;
      char FAR * FAR *  h_addr_list;
};
```

The members of this structure are:

Element	Usage
h_name	Official name of the host (PC).
h_aliases	A NULL-terminated array of alternate names.
h_addrtype	The type of address being returned.
h_length	The length, in bytes, of each address.
h_addr_list	A NULL-terminated list of addresses for the host. Addresses are returned in network byte order.

The macro h_addr is defined to be h_addr_list[0] for compatibility with older software.

The pointer which is returned points to a structure which is allocated by WinSock . The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other WinSock API calls.

h_name is the official name of the host. If using the DNS or similar resolution system, it is the Fully Qualified Domain Name (FQDN) that caused the server to return a reply. If using a local "hosts" file, it is the first entry after the IP address.

Return Value If no error occurs, **gethostbyaddr()** returns a pointer to the hostent structure described above. Otherwise it returns a NULL pointer and a specific error number may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative Answer Host not found.
WSATRY_AGAIN	Non-Authoritative Host not found, or server failed.
WSANO_RECOVERY	Non-recoverable error occurred.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEAFNOSUPPORT	The *type* specified is not supported by the Windows Sockets implementation.
WSAEFAULT	The *addr* argument is not a valid part of the user address space, or the *len* argument is too small.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**

See Also **WSAAsyncGetHostByAddr()**, **gethostbyname()**,

5.4.2. gethostbyname()

Description Get host information corresponding to a hostname.

```
#include <winsock2.h>

struct hostent FAR *
WSAAPI gethostbyname (
        IN      const char FAR *        name
);
```

name A pointer to the null terminated name of the host.

Remarks **gethostbyname()** returns a pointer to a hostent structure as described under **gethostbyaddr()**. The contents of this structure correspond to the hostname *name*.

The pointer which is returned points to a structure which is allocated by WinSock. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other WinSock API calls.

gethostbyname() does not resolve IP address strings passed to it. Such a request is treated exactly as if an unknown host name were passed. An application with an IP address string to resolve should use **inet_addr()** to convert the string to an IP address, then **gethostbyaddr()** to obtain the hostent structure.

gethostbyname() will resolve the string returned by a successful call to **gethostname()**.

Return Value If no error occurs, **gethostbyname()** returns a pointer to the hostent structure described above. Otherwise it returns a NULL pointer and a specific error number may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative Answer Host not found.
WSATRY_AGAIN	Non-Authoritative Host not found, or server failure.
WSANO_RECOVERY	Non-recoverable error occurred.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *name* argument is not a valid part of the user address space.

WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**

See Also **WSAAsyncGetHostByName()**, **gethostbyaddr()**

5.4.3. gethostname()

Description Return the standard host name for the local machine.

```
#include <winsock2.h>

int WSAAPI
gethostname (
        OUT   char FAR *        name,
        IN    int               namelen
);
```

name A pointer to a buffer that will receive the host name.

namelen The length of the buffer.

Remarks This routine returns the name of the local host into the buffer specified by the *name* parameter. The host name is returned as a null-terminated string. The form of the host name is dependent on the WinSock provider--it may be a simple host name, or it may be a fully qualified domain name. However, it is guaranteed that the name returned will be successfully parsed by **gethostbyname()** and **WSAAsyncGetHostByName().**

Note: If no local host name has been configured **gethostname()** must succeed and return a token host name that **gethostbyname()** or **WSAAsyncGetHostByName()** can resolve.

Return Value If no error occurs, **gethostname()** returns 0, otherwise it returns SOCKET_ERROR and a specific error code may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSAEFAULT	The *name* argument is not a valid part of the user address space, or the buffer size specified by *namelen* argument is too small to hold the complete host name.
WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.

See Also **gethostbyname(), WSAAsyncGetHostByName().**

5.4.4. getprotobyname()

Description Get protocol information corresponding to a protocol name.

```
#include <winsock2.h>

struct protoent FAR * WSAAPI
getprotobyname (
        IN      const char FAR *        name
);
```

name A pointer to a null terminated protocol name.

Remarks **getprotobyname()** returns a pointer to the following structure which contains the name(s) and protocol number which correspond to the given protocol *name*. All strings are null terminated.

```
struct protoent {
      char FAR *  p_name;
      char FAR * FAR *  p_aliases;
      short p_proto;
};
```

The members of this structure are:

Element	Usage
p_name	Official name of the protocol.
p_aliases	A NULL-terminated array of alternate names.
p_proto	The protocol number, in host byte order.

The pointer which is returned points to a structure which is allocated by the WinSock library. The application must never attempt to modify this structure or to free any of its components. Furthermore only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other WinSock API calls.

Return Value If no error occurs, **getprotobyname()** returns a pointer to the protoent structure described above. Otherwise it returns a NULL pointer and a specific error number may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative Answer Protocol not found.
WSATRY_AGAIN	Non-Authoritative Protocol not found, or server failure .
WSANO_RECOVERY	Non-recoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *name* argument is not a valid part of the user address space.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**

See Also **WSAAsyncGetProtoByName()**, **getprotobynumber()**

5.4.5. getprotobynumber()

Description Get protocol information corresponding to a protocol number.

```
#include <winsock2.h>

struct protoent FAR * WSAAPI
getprotobynumber (
        IN      int             number
);
```

number A protocol number, in host byte order.

Remarks This function returns a pointer to a protoent structure as described above in **getprotobyname()**. The contents of the structure correspond to the given protocol number.

The pointer which is returned points to a structure which is allocated by WinSock. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other WinSock API calls.

Return Value If no error occurs, **getprotobynumber()** returns a pointer to the protoent structure described above. Otherwise it returns a NULL pointer and a specific error number may be retrieved by calling **WSAGetLastError()**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative Answer Protocol not found.
WSATRY_AGAIN	Non-Authoritative Protocol not found, or server failure .
WSANO_RECOVERY	Non-recoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**

See Also **WSAAsyncGetProtoByNumber()**, **getprotobyname()**

5.4.6. getservbyname()

Description Get service information corresponding to a service name and protocol.

```
#include <winsock2.h>

struct servent FAR * WSAAPI
getservbyname (
        IN      const char FAR*         name,
        IN      const char FAR *        proto
);
```

name A pointer to a null terminated service name.

proto An optional pointer to a null terminated protocol name. If this pointer is NULL, **getservbyname()** returns the first service entry for which the *name* matches the s_name or one of the s_aliases. Otherwise **getservbyname()** matches both the *name* and the *proto*.

Remarks **getservbyname()** returns a pointer to the following structure which contains the name(s) and service number which correspond to the given service *name*. All strings are null terminated.

```
struct servent {
      char FAR *  s_name;
      char FAR * FAR *  s_aliases;
      short s_port;
      char FAR *  s_proto;
};
```

The members of this structure are:

Element	Usage
s_name	Official name of the service.
s_aliases	A NULL-terminated array of alternate names.
s_port	The port number at which the service may be contacted. Port numbers are returned in network byte order.
s_proto	The name of the protocol to use when contacting the service.

The pointer which is returned points to a structure which is allocated by the WinSock library. The application must never attempt to modify this structure or to free any of its components. Furthermore only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other WinSock API calls.

Return Value If no error occurs, **getservbyname()** returns a pointer to the servent structure described above. Otherwise it returns a NULL pointer and a specific error number may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.

WSAHOST_NOT_FOUND	Authoritative Answer Service not found.
WSATRY_AGAIN	Non-Authoritative Service not found, or server failure .
WSANO_RECOVERY	Non-recoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**

See Also **WSAAsyncGetServByName()**, **getservbyport()**

5.4.7. getservbyport()

Description Get service information corresponding to a port and protocol.

```
#include <winsock2.h>

struct servent FAR * WSAAPI
getservbyport (
        IN      int                 port,
        IN      const char FAR*     proto
);
```

port The port for a service, in network byte order.

proto An optional pointer to a protocol name. If this is NULL, **getservbyport()** returns the first service entry for which the *port* matches the s_port. Otherwise **getservbyport()** matches both the *port* and the *proto*.

Remarks **getservbyport()** returns a pointer to a servent structure as described above for **getservbyname()**.

The pointer which is returned points to a structure which is allocated by WinSock. The application must never attempt to modify this structure or to free any of its components. Furthermore, only one copy of this structure is allocated per thread, and so the application should copy any information which it needs before issuing any other WinSock API calls.

Return Value If no error occurs, **getservbyport()** returns a pointer to the servent structure described above. Otherwise it returns a NULL pointer and a specific error number may be retrieved by calling **WSAGetLastError().**

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAHOST_NOT_FOUND	Authoritative Answer Service not found.
WSATRY_AGAIN	Non-Authoritative Service not found, or server failure .
WSANO_RECOVERY	Non-recoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.

WSAEFAULT	The *proto* argument is not a valid part of the user address space.
WSAEINTR	A blocking WinSock 1.1 call was canceled via **WSACancelBlockingCall().**

See Also **WSAAsyncGetServByPort()**, **getservbyname()**

5.4.8. inet_addr()

Description Convert a string containing an (IPv4) Internet Protocol dotted address into an **in_addr**.

```
#include <winsock2.h>

unsigned long WSAAPI
inet_addr (
        IN      const char FAR *        cp
);
```

cp A null terminated character string representing a number expressed in the Internet standard "." notation.

Remarks This function interprets the character string specified by the *cp* parameter. This string represents a numeric Internet address expressed in the Internet standard "." notation. The value returned is a number suitable for use as an Internet address. All Internet addresses are returned in IP's network order (bytes ordered from left to right).

Internet Addresses

Values specified using the "." notation take one of the following forms:

a.b.c.d a.b.c a.b a

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address. Note that when an Internet address is viewed as a 32-bit integer quantity on the Intel architecture, the bytes referred to above appear as "d.c.b.a". That is, the bytes on an Intel processor are ordered from right to left.

Note: The following notations are only used by Berkeley, and nowhere else on the Internet. In the interests of compatibility with their software, they are supported as specified.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host".

When a two part address is specified, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as "net.host".

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

Return Value If no error occurs, **inet_addr()** returns an unsigned long containing a suitable binary representation of the Internet address given. If the passed-in string does not contain a legitimate Internet address, for example if a portion of an "a.b.c.d" address exceeds 255, **inet_addr()** returns the value INADDR_NONE.

See Also **inet_ntoa()**

5.4.9. inet_ntoa()

Description Convert an (IPv4) Internet network address into a string in dotted format.

```
#include <winsock2.h>

char FAR * WSAAPI
inet_ntoa (
        IN      struct in_addr          in
);
```

in A structure which represents an Internet host address.

Remarks This function takes an Internet address structure specified by the *in* parameter. It returns an ASCII string representing the address in "." notation as "a.b.c.d". Note that the string returned by **inet_ntoa()** resides in memory which is allocated by WinSock . The application should not make any assumptions about the way in which the memory is allocated. The data is guaranteed to be valid until the next WinSock API call within the same thread, but no longer.

Return Value If no error occurs, **inet_ntoa()** returns a char pointer to a static buffer containing the text address in standard "." notation. Otherwise, it returns NULL. The data should be copied before another WinSock call is made.

See Also **inet_addr()**.

5.4.10. WSAAsyncGetHostByAddr()

Description Get host information corresponding to an address - asynchronous version.

```
#include <winsock2.h>

HANDLE WSAAPI
WSAAsyncGetHostByAddr (
        IN      HWND                hWnd,
        IN      unsigned int        wMsg,
        IN      const char FAR *    addr,
        IN      int                 len,
        IN      int                 type,
        OUT     char FAR*           buf,
        IN      int                 buflen
);
```

hWnd The handle of the window which should receive a message when the asynchronous request completes.

wMsg The message to be received when the asynchronous request completes.

addr A pointer to the network address for the host. Host addresses are stored in network byte order.

len The length of the address.

type The type of the address.

buf A pointer to the data area to receive the hostent data. Note that this must be larger than the size of a hostent structure. This is because the data area supplied is used by WinSock to contain not only a hostent structure but any and all of the data which is referenced by members of the hostent structure. It is recommended that you supply a buffer of MAXGETHOSTSTRUCT bytes.

buflen The size of data area *buf* above.

Remarks This function is an asynchronous version of **gethostbyaddr()**, and is used to retrieve host name and address information corresponding to a network address. WinSock initiates the operation and returns to the caller immediately, passing back an opaque "asynchronous task handle" which the application may use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation is complete the application's window *hWnd* receives message *wMsg*. The *wParam* argument contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code may be any error as defined in **winsock2.h**. An error code of zero indicates successful completion of the asynchronous operation. On successful completion, the buffer supplied to the original function call contains a hostent structure.

To access the elements of this structure, the original buffer address should be cast to a hostent structure pointer and accessed as appropriate.

Note that if the error code is WSAENOBUFS, it indicates that the size of the buffer specified by *buflen* in the original call was too small to contain all the resultant information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply ALL the requisite information. If the application decides that the partial data is inadequate, it may reissue the **WSAAsyncGetHostByAddr()** function call with a buffer large enough to receive all the desired information (i.e. no smaller than the low 16 bits of *lParam*).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLEN, defined in **winsock2.h** as:

```
#define WSAGETASYNCERROR(lParam)            HIWORD(lParam)
#define WSAGETASYNCBUFLEN(lParam)           LOWORD(lParam)
```

The use of these macros will maximize the portability of the source code for the application.

Return Value The return value specifies whether or not the asynchronous operation was successfully initiated. Note that it does not imply success or failure of the operation itself.

If the operation was successfully initiated, **WSAAsyncGetHostByAddr()** returns a nonzero value of type HANDLE which is the asynchronous task handle (not to be confused with a Windows HTASK) for the request. This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest()**. It can also be used to match up asynchronous operations and completion messages, by examining the *wParam* message argument.

If the asynchronous operation could not be initiated, **WSAAsyncGetHostByAddr()** returns a zero value, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Comments The buffer supplied to this function is used by WinSock to construct a hostent structure together with the contents of data areas referenced by members of the same hostent structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in **winsock2.h**).

Error Codes The following error codes may be set when an application window receives a message. As described above, they may be extracted from the *lParam* in the reply message using the WSAGETASYNCERROR macro.

WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	*addr* or *buf* is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative Answer Host not found.

WSATRY_AGAIN	Non-Authoritative Host not found, or SERVERFAIL.
WSANO_RECOVERY	Non-recoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors may occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the WinSock implementation.

See Also **gethostbyaddr()**, **WSACancelAsyncRequest()**

5.4.11. WSAAsyncGetHostByName()

Description Get host information corresponding to a hostname - asynchronous version.

```
#include <winsock2.h>

HANDLE WSAAPI
WSAAsyncGetHostByName (
        IN      HWND                hWnd,
        IN      unsigned int        wMsg,
        IN      const char FAR *    name,
        OUT     char FAR *          buf,
        IN      int                 buflen
);
```

hWnd The handle of the window which should receive a message when the asynchronous request completes.

wMsg The message to be received when the asynchronous request completes.

name A pointer to the null terminated name of the host.

buf A pointer to the data area to receive the hostent data. Note that this must be larger than the size of a hostent structure. This is because the data area supplied is used by WinSock to contain not only a hostent structure but any and all of the data which is referenced by members of the hostent structure. It is recommended that you supply a buffer of MAXGETHOSTSTRUCT bytes.

buflen The size of data area *buf* above.

Remarks This function is an asynchronous version of **gethostbyname()**, and is used to retrieve host name and address information corresponding to a hostname. WinSock initiates the operation and returns to the caller immediately, passing back an opaque "asynchronous task handle" which the application may use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation is complete the application's window *hWnd* receives message *wMsg*. The *wParam* argument contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code may be any error as defined in **winsock2.h**. An error code of zero indicates successful completion of the asynchronous operation. On successful completion, the buffer supplied to the original function call contains a hostent structure. To access the elements of this structure, the original buffer address should be cast to a hostent structure pointer and accessed as appropriate.

Note that if the error code is WSAENOBUFS, it indicates that the size of the buffer specified by *buflen* in the original call was too small to contain all the resultant information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply ALL the requisite information. If the application decides that the partial data is inadequate, it may reissue the **WSAAsyncGetHostByName()** function call with a

buffer large enough to receive all the desired information (i.e. no smaller than the low 16 bits of *lParam*).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLEN, defined in **winsock2.h** as:

```
#define WSAGETASYNCERROR(lParam)            HIWORD(lParam)
#define WSAGETASYNCBUFLEN(lParam)           LOWORD(lParam)
```

The use of these macros will maximize the portability of the source code for the application.

WSAAsyncGetHostByName()is guaranteed to resolve the string returned by a successful call to **gethostname()**.

Return Value The return value specifies whether or not the asynchronous operation was successfully initiated. Note that it does not imply success or failure of the operation itself.

If the operation was successfully initiated, **WSAAsyncGetHostByName()** returns a nonzero value of type HANDLE which is the asynchronous task handle (not to be confused with a Windows HTASK) for the request. This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest()**. It can also be used to match up asynchronous operations and completion messages, by examining the *wParam* message argument.

If the asynchronous operation could not be initiated, **WSAAsyncGetHostByName()** returns a zero value, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Comments The buffer supplied to this function is used by WinSock to construct a hostent structure together with the contents of data areas referenced by members of the same hostent structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in **winsock2.h**).

Error Codes The following error codes may be set when an application window receives a message. As described above, they may be extracted from the *lParam* in the reply message using the WSAGETASYNCERROR macro.

WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	*name* or *buf* is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative Answer Host not found.
WSATRY_AGAIN	Non-Authoritative Host not found, or SERVERFAIL.
WSANO_RECOVERY	Non-recoverable errors, FORMERR, REFUSED, NOTIMP.

WSANO_DATA	Valid name, no data record of requested type.

The following errors may occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the WinSock implementation.

See Also **gethostbyname()**, **WSACancelAsyncRequest()**

5.4.12. WSAAsyncGetProtoByName()

Description Get protocol information corresponding to a protocol name - asynchronous version.

```
#include <winsock2.h>

HANDLE WSAAPI
WSAAsyncGetProtoByName (
        IN      HWND                hWnd,
        IN      unsigned int        wMsg,
        IN      const char FAR *    name,
        OUT     char FAR *          buf,
        IN      int                 buflen
);
```

hWnd The handle of the window which should receive a message when the asynchronous request completes.

wMsg The message to be received when the asynchronous request completes.

name A pointer to the null terminated protocol name to be resolved.

buf A pointer to the data area to receive the protoent data. Note that this must be larger than the size of a protoent structure. This is because the data area supplied is used by WinSock to contain not only a protoent structure but any and all of the data which is referenced by members of the protoent structure. It is recommended that you supply a buffer of MAXGETHOSTSTRUCT bytes.

buflen The size of data area *buf* above.

Remarks This function is an asynchronous version of **getprotobyname()**, and is used to retrieve the protocol name and number corresponding to a protocol name. WinSock initiates the operation and returns to the caller immediately, passing back an opaque "asynchronous task handle" which the application may use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation is complete the application's window *hWnd* receives message *wMsg*. The *wParam* argument contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code may be any error as defined in **winsock2.h**. An error code of zero indicates successful completion of the asynchronous operation. On successful completion, the buffer supplied to the original function call contains a protoent structure. To access the elements of this structure, the original buffer address should be cast to a protoent structure pointer and accessed as appropriate.

Note that if the error code is WSAENOBUFS, it indicates that the size of the buffer specified by *buflen* in the original call was too small to contain all the resultant information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply ALL the requisite information. If the application decides that the partial data is inadequate, it may reissue the **WSAAsyncGetProtoByName()** function call with a

buffer large enough to receive all the desired information (i.e. no smaller than the low 16 bits of *lParam*).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLEN, defined in **winsock2.h** as:

```
#define WSAGETASYNCERROR(lParam)            HIWORD(lParam)
#define WSAGETASYNCBUFLEN(lParam)           LOWORD(lParam)
```

The use of these macros will maximize the portability of the source code for the application.

Return Value The return value specifies whether or not the asynchronous operation was successfully initiated. Note that it does not imply success or failure of the operation itself.

If the operation was successfully initiated, **WSAAsyncGetProtoByName()** returns a nonzero value of type HANDLE which is the asynchronous task handle for the request (not to be confused with a Windows HTASK). This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest()**. It can also be used to match up asynchronous operations and completion messages, by examining the *wParam* message argument.

If the asynchronous operation could not be initiated, **WSAAsyncGetProtoByName()** returns a zero value, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Comments The buffer supplied to this function is used by WinSock to construct a protoent structure together with the contents of data areas referenced by members of the same protoent structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in **winsock2.h**).

Error Codes The following error codes may be set when an application window receives a message. As described above, they may be extracted from the *lParam* in the reply message using the WSAGETASYNCERROR macro.

WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	*name* or *buf* is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative Answer Protocol not found.
WSATRY_AGAIN	Non-Authoritative Protocol not found, or server failure .
WSANO_RECOVERY	Non-recoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors may occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the WinSock implementation.

See Also **getprotobyname()**, **WSACancelAsyncRequest()**

5.4.13. WSAAsyncGetProtoByNumber()

Description Get protocol information corresponding to a protocol number - asynchronous version.

```
#include <winsock2.h>

HANDLE WSAAPI
WSAAsyncGetProtoByNumber (
        IN      HWND            hWnd,
        IN      unsigned int    wMsg,
        IN      int             number,
        OUT     char FAR *      buf,
        IN      int             buflen
);
```

hWnd The handle of the window which should receive a message when the asynchronous request completes.

wMsg The message to be received when the asynchronous request completes.

number The protocol number to be resolved, in host byte order.

buf A pointer to the data area to receive the protoent data. Note that this must be larger than the size of a protoent structure. This is because the data area supplied is used by WinSock to contain not only a protoent structure but any and all of the data which is referenced by members of the protoent structure. It is recommended that you supply a buffer of MAXGETHOSTSTRUCT bytes.

buflen The size of data area *buf* above.

Remarks This function is an asynchronous version of **getprotobynumber()**, and is used to retrieve the protocol name and number corresponding to a protocol number. WinSock initiates the operation and returns to the caller immediately, passing back an opaque "asynchronous task handle" which the application may use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation is complete the application's window *hWnd* receives message *wMsg*. The *wParam* argument contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code may be any error as defined in **winsock2.h**. An error code of zero indicates successful completion of the asynchronous operation. On successful completion, the buffer supplied to the original function call contains a protoent structure. To access the elements of this structure, the original buffer address should be cast to a protoent structure pointer and accessed as appropriate.

Note that if the error code is WSAENOBUFS, it indicates that the size of the buffer specified by *buflen* in the original call was too small to contain all the resultant information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply ALL the requisite information. If the application decides that the partial data is inadequate, it may reissue the **WSAAsyncGetProtoByNumber()** function call with a

buffer large enough to receive all the desired information (i.e. no smaller than the low 16 bits of *lParam*).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLEN, defined in **winsock2.h** as:

```
#define WSAGETASYNCERROR(lParam)            HIWORD(lParam)
#define WSAGETASYNCBUFLEN(lParam)           LOWORD(lParam)
```

The use of these macros will maximize the portability of the source code for the application.

Return Value The return value specifies whether or not the asynchronous operation was successfully initiated. Note that it does not imply success or failure of the operation itself.

If the operation was successfully initiated, **WSAAsyncGetProtoByNumber()** returns a nonzero value of type HANDLE which is the asynchronous task handle for the request (not to be confused with a Windows HTASK). This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest()**. It can also be used to match up asynchronous operations and completion messages, by examining the *wParam* message argument.

If the asynchronous operation could not be initiated, **WSAAsyncGetProtoByNumber()** returns a zero value, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Comments The buffer supplied to this function is used by WinSock to construct a protoent structure together with the contents of data areas referenced by members of the same protoent structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in **winsock2.h**).

Error Codes The following error codes may be set when an application window receives a message. As described above, they may be extracted from the *lParam* in the reply message using the WSAGETASYNCERROR macro.

WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	*buf* is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative Answer Protocol not found.
WSATRY_AGAIN	Non-Authoritative Protocol not found, or server failure .
WSANO_RECOVERY	Non-recoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors may occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the WinSock implementation.

See Also **getprotobynumber()**, **WSACancelAsyncRequest()**

5.4.14. WSAAsyncGetServByName()

Description Get service information corresponding to a service name and port - asynchronous version.

```
#include <winsock2.h>

HANDLE WSAAPI
WSAAsyncGetServByName (
        IN      HWND                hWnd,
        IN      unsigned int        wMsg,
        IN      const char FAR *    name,
        IN      const char FAR *    proto,
        OUT     char FAR *          buf,
        IN      int                 buflen
);
```

hWnd The handle of the window which should receive a message when the asynchronous request completes.

wMsg The message to be received when the asynchronous request completes.

name A pointer to a null terminated service name.

proto A pointer to a protocol name. This may be NULL, in which case **WSAAsyncGetServByName()** will search for the first service entry for which *s_name* or one of the *s_aliases* matches the given *name*. Otherwise **WSAAsyncGetServByName()** matches both *name* and *proto*.

buf A pointer to the data area to receive the servent data. Note that this must be larger than the size of a servent structure. This is because the data area supplied is used by WinSock to contain not only a servent structure but any and all of the data which is referenced by members of the servent structure. It is recommended that you supply a buffer of MAXGETHOSTSTRUCT bytes.

buflen The size of data area *buf* above.

Remarks This function is an asynchronous version of **getservbyname()**, and is used to retrieve service information corresponding to a service name. WinSock initiates the operation and returns to the caller immediately, passing back an opaque "asynchronous task handle" which the application may use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation is complete the application's window *hWnd* receives message *wMsg*. The *wParam* argument contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code may be any error as defined in **winsock2.h**. An error code of zero indicates successful completion of the asynchronous operation. On successful completion, the buffer supplied to the original function call contains a servent structure.

To access the elements of this structure, the original buffer address should be cast to a servent structure pointer and accessed as appropriate.

Note that if the error code is WSAENOBUFS, it indicates that the size of the buffer specified by *buflen* in the original call was too small to contain all the resultant information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply ALL the requisite information. If the application decides that the partial data is inadequate, it may reissue the **WSAAsyncGetServByName()** function call with a buffer large enough to receive all the desired information (i.e. no smaller than the low 16 bits of *lParam*).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLEN, defined in **winsock2.h** as:

```
#define WSAGETASYNCERROR(lParam)            HIWORD(lParam)
#define WSAGETASYNCBUFLEN(lParam)           LOWORD(lParam)
```

The use of these macros will maximize the portability of the source code for the application.

Return Value The return value specifies whether or not the asynchronous operation was successfully initiated. Note that it does not imply success or failure of the operation itself.

If the operation was successfully initiated, **WSAAsyncGetServByName()** returns a nonzero value of type HANDLE which is the asynchronous task handle for the request (not to be confused with a Windows HTASK). This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest()**. It can also be used to match up asynchronous operations and completion messages, by examining the *wParam* message argument.

If the asynchronous operation could not be initiated, **WSAAsyncServByName()** returns a zero value, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Comments The buffer supplied to this function is used by WinSock to construct a servent structure together with the contents of data areas referenced by members of the same servent structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in **winsock2.h**).

Error Codes The following error codes may be set when an application window receives a message. As described above, they may be extracted from the *lParam* in the reply message using the WSAGETASYNCERROR macro.

WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	*buf* is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative Answer Host not found.

WSATRY_AGAIN	Non-Authoritative Service not found, or server failure .
WSANO_RECOVERY	Non-recoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors may occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the WinSock implementation.

See Also **getservbyname()**, **WSACancelAsyncRequest()**

5.4.15. WSAAsyncGetServByPort()

Description Get service information corresponding to a port and protocol - asynchronous version.

```
#include <winsock2.h>

HANDLE WSAAPI
WSAAsyncGetServByPort (
        IN      HWND                hWnd,
        IN      unsigned int        wMsg,
        IN      int                 port,
        IN      const char FAR *    proto,
        OUT     char FAR *          buf,
        IN      int                 buflen
);
```

hWnd The handle of the window which should receive a message when the asynchronous request completes.

wMsg The message to be received when the asynchronous request completes.

port The port for the service, in network byte order.

proto A pointer to a protocol name. This may be NULL, in which case **WSAAsyncGetServByPort()** will search for the first service entry for which *s_port* match the given *port*. Otherwise **WSAAsyncGetServByPort()** matches both *port* and *proto*.

buf A pointer to the data area to receive the servent data. Note that this must be larger than the size of a servent structure. This is because the data area supplied is used by WinSock to contain not only a servent structure but any and all of the data which is referenced by members of the servent structure. It is recommended that you supply a buffer of MAXGETHOSTSTRUCT bytes.

buflen The size of data area *buf* above.

Remarks This function is an asynchronous version of **getservbyport()**, and is used to retrieve service information corresponding to a port number. WinSock initiates the operation and returns to the caller immediately, passing back an opaque "asynchronous task handle" which the application may use to identify the operation. When the operation is completed, the results (if any) are copied into the buffer provided by the caller and a message is sent to the application's window.

When the asynchronous operation is complete the application's window *hWnd* receives message *wMsg*. The *wParam* argument contains the asynchronous task handle as returned by the original function call. The high 16 bits of *lParam* contain any error code. The error code may be any error as defined in **winsock2.h**. An error code of zero indicates successful completion of the asynchronous operation. On successful completion, the buffer supplied to the original function call contains a servent structure. To access the elements of this structure, the original buffer address should be cast to a servent structure pointer and accessed as appropriate.

Note that if the error code is WSAENOBUFS, it indicates that the size of the buffer specified by *buflen* in the original call was too small to contain all the resultant information. In this case, the low 16 bits of *lParam* contain the size of buffer required to supply ALL the requisite information. If the application decides that the partial data is inadequate, it may reissue the **WSAAsyncGetServByPort()** function call with a buffer large enough to receive all the desired information (i.e. no smaller than the low 16 bits of *lParam*).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLEN, defined in **winsock2.h** as:

```
#define WSAGETASYNCERROR(lParam)            HIWORD(lParam)
#define WSAGETASYNCBUFLEN(lParam)           LOWORD(lParam)
```

The use of these macros will maximize the portability of the source code for the application.

Return Value The return value specifies whether or not the asynchronous operation was successfully initiated. Note that it does not imply success or failure of the operation itself.

If the operation was successfully initiated, **WSAAsyncGetServByPort()** returns a nonzero value of type HANDLE which is the asynchronous task handle for the request (not to be confused with a Windows HTASK). This value can be used in two ways. It can be used to cancel the operation using **WSACancelAsyncRequest()**. It can also be used to match up asynchronous operations and completion messages, by examining the *wParam* message argument.

If the asynchronous operation could not be initiated, **WSAAsyncGetServByPort()** returns a zero value, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Comments The buffer supplied to this function is used by WinSock to construct a servent structure together with the contents of data areas referenced by members of the same servent structure. To avoid the WSAENOBUFS error noted above, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in **winsock2.h**).

Error Codes The following error codes may be set when an application window receives a message. As described above, they may be extracted from the *lParam* in the reply message using the WSAGETASYNCERROR macro.

WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	Insufficient buffer space is available.
WSAEFAULT	*proto* or *buf* is not in a valid part of the process address space.
WSAHOST_NOT_FOUND	Authoritative Answer Port not found.
WSATRY_AGAIN	Non-Authoritative Port not found, or server failure .

WSANO_RECOVERY	Non-recoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors may occur at the time of the function call, and indicate that the asynchronous operation could not be initiated.

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEWOULDBLOCK	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the WinSock implementation.

See Also **getservbyport()**, **WSACancelAsyncRequest()**

5.4.16. WSACancelAsyncRequest()

Description Cancel an incomplete asynchronous operation.

```
#include <winsock2.h>

int WSAAPI
WSACancelAsyncRequest (
        IN      HANDLE          hAsyncTaskHandle
);
```

hAsyncTaskHandle Specifies the asynchronous operation to be canceled.

Remarks The **WSACancelAsyncRequest()** function is used to cancel an asynchronous operation which was initiated by one of the **WSAAsyncGetXByY()** functions such as **WSAAsyncGetHostByName()**. The operation to be canceled is identified by the *hAsyncTaskHandle* parameter, which should be set to the asynchronous task handle as returned by the initiating **WSAAsyncGetXByY()** function.

Return Value The value returned by **WSACancelAsyncRequest()** is 0 if the operation was successfully canceled. Otherwise the value SOCKET_ERROR is returned, and a specific error number may be retrieved by calling **WSAGetLastError()**.

Comments An attempt to cancel an existing asynchronous **WSAAsyncGetXByY()** operation can fail with an error code of WSAEALREADY for two reasons. First, the original operation has already completed and the application has dealt with the resultant message. Second, the original operation has already completed but the resultant message is still waiting in the application window queue.

Note It is unclear whether the application can usefully distinguish between WSAEINVAL and WSAEALREADY, since in both cases the error indicates that there is no asynchronous operation in progress with the indicated handle. [Trivial exception: 0 is always an invalid asynchronous task handle.] The WinSock specification does not prescribe how a conformant WinSock provider should distinguish between the two cases. For maximum portability, a WinSock application should treat the two errors as equivalent.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup()** must occur before using this API.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that the specified asynchronous task handle was invalid
WSAEINPROGRESS	A blocking WinSock 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	The asynchronous routine being canceled has already completed.

See Also **WSAAsyncGetHostByAddr()**, **WSAAsyncGetHostByName()**, **WSAAsyncGetProtoByNumber()**, **WSAAsyncGetProtoByName()**, **WSAAsyncGetServByPort()**, **WSAAsyncGetServByName().**

Appendix A. Error Codes and Header Files and Data Types

A.1 Error Codes

A.1.1 Error Codes - Brief Description

The following is a list of possible error codes returned by the **WSAGetLastError()** call, along with their brief explanations. The error numbers are consistently set across all WinSock-compliant implementations.

WinSock code	Berkeley equivalent	Error	Interpretation
WSAEINTR	EINTR	10004	As in standard C
WSAEBADF	EBADF	10009	As in standard C
WSAEACCES	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 any WinSock 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 WinSock 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
		Missing 10066 thru 10071	
WSASYSNOTREADY		10091	Returned by **WSAStartup()** indicating that the network subsystem is unusable.
WSAVERNOTSUPPORTED		10092	Returned by **WSAStartup()** indicating that the WinSock DLL cannot support this app.
WSANOTINITIALISED		10093	Returned by any function except **WSAStartup()** indicating that a successful **WSAStartup()** has not yet been performed.
WSAEDISCON		10010 1	Returned by **WSARecv()**, **WSARecvFrom()** to indicate the remote party has initiated a graceful shutdown sequence.
		Missing 10102 thru 10112	
WSA_OPERATION_ABORTED		*	An overlapped operation has been canceled due to the closure of the socket, or the execution of the SIO_FLUSH command in **WSAIoctl()**
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

* varies in different operating systems

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 WinSock versions of regular Berkeley Sockets error codes.

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

The fourth set of errors are returned by WinSock **getXbyY()** and **WSAAsyncGetXByY()** functions, 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 WinSock provider does not use the DNS, it will use the most appropriate code. In general, a WinSock 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 **Winsock2.h** header file listed in section A.2.2 , and are based on the fact that WinSock 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 **Winsock2.h**. This is because it includes only errors which might reasonably be returned by a WinSock implementation: **Winsock2.h**, on the other hand, includes a full set of BSD definitions to ensure compatibility with ported software.

A.1.2 Error Codes - Extended Description

The following is a list of possible error codes returned by the **WSAGetLastError()** call, along with their extended explanations. Errors are listed in alphabetical order by error macro. Some error codes defined in WINSOCK2.H are not returned from any function - these have not been listed here.

WSAEACCES **(10013)** *Permission denied.*

An attempt was made to access a socket in a way forbidden by its access permissions. An example is using a broadcast address for **sendto()** without broadcast permission being set using **setsockopt(SO_BROADCAST).**

WSAEADDRINUSE **(10048)** *Address already in use.*

Only one usage of each socket address (protocol/IP address/port) is normally permitted. This error occurs if an application attempts to **bind()** a socket to an IP address/port that has already been used for an existing socket, or a socket that wasn't closed properly, or one that is still in the process of closing. For server applications that need to **bind()** multiple sockets to the same port number, consider using **setsockopt(SO_REUSEADDR)**. Client applications usually need not call **bind()** at all - **connect()** will choose an unused port automatically. When the **bind()** is done to a wild-card address (involving ADDR_ANY), a WSAEADDRINUSE error could be delayed until the specific address is “committed”. This could happen in a later function such as **connect()**, **listen()**, **WSAConnect()**, or **WSAJoinLeaf()**.

WSAEADDRNOTAVAIL **(10049)** *Cannot assign requested address.*

The requested address is not valid in its context. Normally results from an attempt to **bind()** to an address that is not valid for the local machine. This may also result from **connect(), sendto(), WSAConnect(), WSAJoinLeaf(),** or **WSASendTo()** when the remote address or port is not valid for a remote machine (e.g. address or port 0).

WSAEAFNOSUPPORT (10047) *Address family not supported by protocol family*.
An address incompatible with the requested protocol was used. All sockets are created with an associated "address family" (i.e. AF_INET for Internet Protocols) and a generic protocol type (i.e. SOCK_STREAM). This error will be returned if an incorrect protocol is explicitly requested in the **socket()** call, or if an address of the wrong family is used for a socket, e.g. in **sendto()**.

WSAEALREADY (10037) *Operation already in progress.*
An operation was attempted on a non-blocking socket that already had an operation in progress - i.e. calling **connect()** a second time on a non-blocking socket that is already connecting, or canceling an asynchronous request (**WSAAsyncGetXbyY()**) that has already been canceled or completed.

WSAECONNABORTED (10053) *Software caused connection abort*.
An established connection was aborted by the software in your host machine, possibly due to a data transmission timeout or protocol error.

WSAECONNREFUSED (10061) *Connection refused*.
No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host - i.e. one with no server application running.

WSAECONNRESET (10054) *Connection reset by peer*.
A existing connection was forcibly closed by the remote host. This normally results if the peer application on the remote host is suddenly stopped, the host is rebooted, or the remote host used a "hard close" (see **setsockopt(SO_LINGER)**) on the remote socket. This error may also result if a connection was broken due to “keep-alive” activity detecting a failure while one or more operations are in progress. Operations that were in progress fail with WSAENETRESET. Subsequent operations fail with WSAECONNRESET.

WSAEDESTADDRREQ (10039) *Destination address required.*
A required address was omitted from an operation on a socket. For example, this error will be returned if **sendto()** is called with the remote address of ADDR_ANY.

WSAEFAULT (10014) *Bad address.*
The system detected an invalid pointer address in attempting to use a pointer argument of a call. This error occurs if an application passes an invalid pointer value, or if the length of the buffer is too small. For instance, if the length of an argument which is a struct sockaddr is smaller than sizeof(struct sockaddr).

WSAEHOSTDOWN (10064) *Host is down.*
A socket operation failed because the destination host was down. A socket operation encountered a dead host. Networking activity on the local host has not been initiated. These conditions are more likely to be indicated by the error WSAETIMEDOUT.

WSAEHOSTUNREACH (10065) *No route to host*.
A socket operation was attempted to an unreachable host. See WSAENETUNREACH

WSAEINPROGRESS (10036) *Operation now in progress.*
A blocking operation is currently executing. Windows Sockets only allows a single blocking operation to be outstanding per task (or thread), and if any other function call is made (whether or not it references that or any other socket) the function fails with the WSAEINPROGRESS error.

WSAEINTR (10004) *Interrupted function call.*

A blocking operation was interrupted by a call to **WSACancelBlockingCall()**.

WSAEINVAL **(10022)** *Invalid argument.*
Some invalid argument was supplied (for example, specifying an invalid level to the **setsockopt()** function). In some instances, it also refers to the current state of the socket - for instance, calling **accept()** on a socket that is not **listen()**ing.

WSAEISCONN **(10056)** *Socket is already connected.*
A connect request was made on an already connected socket. Some implementations also return this error if **sendto()** is called on a connected SOCK_DGRAM socket (For SOCK_STREAM sockets, the *to* parameter in **sendto()** is ignored), although other implementations treat this as a legal occurrence.

WSAEMFILE **(10024)** *Too many open files.*
Too many open sockets. Each implementation may have a maximum number of socket handles available, either globally, per process or per thread.

WSAEMSGSIZE **(10040)** *Message too long.*
A message sent on a datagram socket was larger than the internal message buffer or some other network limit, or the buffer used to receive a datagram into was smaller than the datagram itself.

WSAENETDOWN **(10050)** *Network is down.*
A socket operation encountered a dead network. This could indicate a serious failure of the network system (i.e. the protocol stack that the WinSock DLL runs over), the network interface, or the local network itself.

WSAENETRESET **(10052)** *Network dropped connection on reset.*
The connection has been broken due to "keep-alive" activity detecting a failure while the operation was in progress. May also be returned by **setsockopt()** if an attempt is made to set SO_KEEPALIVE on a connection that has already failed.

WSAENETUNREACH **(10051)** *Network is unreachable.*
A socket operation was attempted to an unreachable network. This usually means the local software knows no route to reach the remote host.

WSAENOBUFS **(10055)** *No buffer space available.*
An operation on a socket could not be performed because the system lacked sufficient buffer space or because a queue was full.

WSAENOPROTOOPT **(10042)** *Bad protocol option.*
An unknown, invalid or unsupported option or level was specified in a **getsockopt()** or **setsockopt()** call.

WSAENOTCONN **(10057)** *Socket is not connected.*
A request to send or receive data was disallowed because the socket is not connected and (when sending on a datagram socket using **sendto()**) no address was supplied. Any other type of operation might also return this error - for example, **setsockopt()** setting SO_KEEPALIVE if the connection has been reset.

WSAENOTSOCK **(10038)** *Socket operation on non-socket.*
An operation was attempted on something that is not a socket. Either the socket handle parameter did not reference a valid socket, or for **select()**, a member of an fd_set was not valid.

WSAEOPNOTSUPP **(10045)** *Operation not supported.*
The attempted operation is not supported for the type of object referenced. Usually this occurs when a socket descriptor to a socket that cannot support this operation, for example, trying to accept a connection on a datagram socket.

WSAEPFNOSUPPORT **(10046)** *Protocol family not supported.*
The protocol family has not been configured into the system or no implementation for it exists. Has a slightly different meaning to WSAEAFNOSUPPORT, but is interchangeable in most cases, and all Windows Sockets functions that return one of these specify WSAEAFNOSUPPORT.

WSAEPROCLIM **(10067)** *Too many processes.*
A Windows Sockets implementation may have a limit on the number of applications that may use it simultaneously. **WSAStartup()** may fail with this error if the limit has been reached.

WSAEPROTONOSUPPORT **(10043)** *Protocol not supported.*
The requested protocol has not been configured into the system, or no implementation for it exists. For example, a **socket()** call requests a SOCK_DGRAM socket, but specifies a stream protocol.

WSAEPROTOTYPE **(10041)** *Protocol wrong type for socket.*
A protocol was specified in the **socket()** function call that does not support the semantics of the socket type requested. For example, the ARPA Internet UDP protocol cannot be specified with a socket type of SOCK_STREAM.

WSAESHUTDOWN **(10058)** *Cannot send after socket shutdown.*
A request to send or receive data was disallowed because the socket had already been shut down in that direction with a previous **shutdown()** call. By calling **shutdown()** a partial close of a socket is requested, which is a signal that sending or receiving or both has been discontinued.

WSAESOCKTNOSUPPORT **(10044)** *Socket type not supported.*
The support for the specified socket type does not exist in this address family. For example, the optional type SOCK_RAW might be selected in a **socket()** call, and the implementation does not support SOCK_RAW sockets at all.

WSAETIMEDOUT **(10060)** *Connection timed out.*
A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond.

WSATYPE_NOT_FOUND **(10109)** *Class type not found*
The specified class was not found.

WSAEWOULDBLOCK **(10035)** *Resource temporarily unavailable.*
This error is returned from operations on non-blocking sockets that cannot be completed immediately, for example **recv()** when no data is queued to be read from the socket. It is a non-fatal error, and the operation should be retried later. It is normal for WSAEWOULDBLOCK to be reported as the result from calling **connect()** on a non-blocking SOCK_STREAM socket, since some time must elapse for the connection to be established.

WSAHOST_NOT_FOUND **(11001)** *Host not found.*
No such host is known. The name is not an official hostname or alias, or it cannot be found in the database(s) being queried. This error may also be returned for protocol and service queries, and means the specified name could not be found in the relevant database.

WSA_INVALID_HANDLE **(OS dependent)** *Specified event object handle is invalid.*
An application attempts to use an event object, but the specified handle is not valid.

WSA_INVALID_PARAMETER **(OS dependent)** *One or more parameters are invalid.*
An application used a WinSock function which directly maps to a Win32 function. The Win32 function is indicating a problem with one or more parameters.

WSAINVALIDPROCTABLE **(OS dependent)** *Invalid procedure table from service provider.*
A service provider returned a bogus proc table to WS2_32.DLL. (Usually caused by one or more of the function pointers being NULL.)

WSAINVALIDPROVIDER **(OS dependent)** *Invalid service provider version number.*
A service provider returned a version number other than 2.2.

WSA_IO_INCOMPLETE **(OS dependent)** *Overlapped I/O event object not in signaled state.*
The application has tried to determine the status of an overlapped operation which is not yet completed. Applications that use **WSAGetOverlappedResult()** (with the *fWait* flag set to false) in a polling mode to determine when an overlapped operation has completed will get this error code until the operation is complete.

WSA_IO_PENDING **(OS dependent)** *Overlapped operations will complete later.*
The application has initiated an overlapped operation which cannot be completed immediately. A completion indication will be given at a later time when the operation has been completed.

WSA_NOT_ENOUGH_MEMORY **(OS dependent)** *Insufficient memory available.*
An application used a WinSock function which directly maps to a Win32 function. The Win32 function is indicating a lack of required memory resources.

WSANOTINITIALISED **(10093)** *Successful WSAStartup() not yet performed.*
Either the application hasn't called **WSAStartup()**, or **WSAStartup()** failed. The application may be accessing a socket which the current active task does not own (i.e. trying to share a socket between tasks), or **WSACleanup()** has been called too many times.

WSANO_DATA **(11004)** *Valid name, no data record of requested type.*
The requested name is valid and was found in the database, but it does not have the correct associated data being resolved for. The usual example for this is a hostname -> address translation attempt (using **gethostbyname()** or **WSAAsyncGetHostByName()**) which uses the DNS (Domain Name Server), and an MX record is returned but no A record - indicating the host itself exists, but is not directly reachable.

WSANO_RECOVERY **(11003)** *This is a non-recoverable error.*
This indicates some sort of non-recoverable error occurred during a database lookup. This may be because the database files (e.g. BSD-compatible HOSTS, SERVICES or PROTOCOLS files) could not be found, or a DNS request was returned by the server with a severe error.

WSAPROVIDERFAILEDINIT **(OS dependent)** *Unable to initialize a service provider.*
Either a service provider's DLL could not be loaded (**LoadLibrary()** failed) or the provider's WSPStartup/NSPStartup function failed.

WSASYSCALLFAILURE **(OS dependent)** *System call failure..*
Returned when a system call that should never fail does. For example, if a call to **WaitForMultipleObjects()** fails or one of the registry APIs fails trying to manipulate the protocol/namespace catalogs.

WSASYSNOTREADY (10091) *Network subsystem is unavailable.*

This error is returned by **WSAStartup()** if the Windows Sockets implementation cannot function at this time because the underlying system it uses to provide network services is currently unavailable. Users should check:

- that the appropriate Windows Sockets DLL file is in the current path,
- • that they are not trying to use more than one WinSock implementation simultaneously. If there is more than one WINSOCK DLL on your system, be sure the first one in the path is appropriate for the network subsystem currently loaded.
- the WinSock implementation documentation to be sure all necessary components are currently installed and configured correctly.

WSATRY_AGAIN (11002) *Non-authoritative host not found.*

This is usually a temporary error during hostname resolution and means that the local server did not receive a response from an authoritative server. A retry at some time later may be successful.

WSAVERNOTSUPPORTED (10092) *WINSOCK.DLL version out of range.*

The current WinSock implementation does not support the Windows Sockets specification version requested by the application. Check that no old Windows Sockets DLL files are being accessed.

WSAEDISCON (10101) *Graceful shutdown in progress.*

Returned by **WSARecv()**, **WSARecvFrom()** to indicate the remote party has initiated a graceful shutdown sequence.

WSA_OPERATION_ABORTED (OS dependent) *Overlapped operation aborted.*

An overlapped operation was canceled due to the closure of the socket, or the execution of the SIO_FLUSH command in **WSAIoctl()**

A.2 Header Files

A.2.1 Berkeley Header Files

The WinSock SDK includes a set of vestigial header files with names that match a number of the header files in the Berkeley software distribution. These files are provided for source code compatibility only, and each consists of three lines:

```
#ifndef _WinsockAPI_
#include <Winsock2.h>
#endif
```

The header files provided for compatibility are:
netdb.h
arpa/inet.h
sys/time.h
sys/socket.h
netinet/in.h

The file **Winsock2.h** contains all of the type and structure definitions, constants, macros, and function prototypes used by the WinSock specification. An application writer may choose to ignore the compatibility headers and include **Winsock2.h** in each source file.

A.2.2 WinSock Header File - Winsock2.h

The **Winsock2.h** header file includes a number of types and definitions from the standard Windows header file **windows.h**.

A WinSock service provider vendor **MUST NOT** make any modifications to this header file which could impact binary compatibility of WinSock applications. The constant values, function parameters and return codes, and the like must remain consistent across all WinSock service provider vendors.

New versions of **Winsock2.h** will appear periodically as new identifiers are allocated by the WinSock Identifier Clearinghouse. The clearinghouse can be reached via the world wide web at

```
http://www.stardust.com/wsresource/winsock2/ws2ident.html
```

Developers are urged to stay current with successive revisions of Winsock2.h as they are made available by the clearinghouse.

The **Winsock2.h** header file now supports UNICODE, and thus contains both A and W declarations for all applicable functions and structures. In addition, both function prototypes and function typedefs are supplied. As a result, the **Winsock2.h** header file has become quite lengthy (in excess of 40 pages when printed). Because it has grown so large and is subject to frequent updates, **Winsock2.h** is no longer being copied verbatim into this specification document.

A.2.3 Sizes of Data Types

This section lists the primitive data types used as parameters and return values for the Windows Sockets API functions, specifying their sizes in bytes. This is to ensure that developers using programming environments and languages other than C/C++ will be able to develop modules that will be capable of correctly interfacing with a Windows Sockets implementation's WinSock DLL.

Primitive Data Type	16-bit Windows data sizes (in bytes)	32-bit Windows data sizes (in bytes)
BOOL	2	4
char	1	1
int	2	4
FAR pointer to anything	4	4
long	4	4
short	2	2
SOCKET	2	4

Appendix B. Multipoint and Multicast Semantics

B.1. Multipoint and Multicast Introduction

In considering how to support multipoint and multicast in WinSock 2 a number of existing and proposed multipoint/multicast schemes (including IP-multicast, ATM point-to-multipoint connection, ST-II, T.120, H.320 (MCU), etc.) were examined. While common in some aspects, each is widely different in others. To enable a coherent discussion of the various schemes, it is valuable to first create a taxonomy that characterizes the essential attributes of each. For simplicity, the term "multipoint" will hereafter be used to represent both multipoint and multicast.

B.2 Multipoint Taxonomy

The taxonomy described in this appendix first distinguishes the control plane that concerns itself with the way a multipoint session is established, from the data plane that deals with the transfer of data amongst session participants.

In the control plane there are two distinct types of session establishment: *rooted* and *non-rooted*. In the case of rooted control, there exists a special participant, called c_root, that is different from the rest of the members of this multipoint session, each of which is called a c_leaf. The c_root must remain present for the whole duration of the multipoint session, as the session will be broken up in the absence of the c_root. The c_root usually initiates the multipoint session by setting up the connection to a c_leaf, or a number of c_leafs. The c_root may add more c_leafs, or (in some cases) a c_leaf can join the c_root at a later time. Examples of the rooted control plane can be found in ATM and ST-II.

For a non-rooted control plane, all the members belonging to a multipoint session are leaves, i.e., no special participant acting as a c_root exists. Each c_leaf needs to add itself to a pre-existing multipoint session that either is always available (as in the case of an IP multicast address), or has been set up through some out-of-band mechanism which is outside the scope of the WinSock specification. Another way to look at this is that a c_root still exists, but can be considered to be in the network cloud as opposed to one of the participants. Because a control root still exists, a non-rooted control plane could also be considered to be implicitly rooted. Examples for this kind of implicitly rooted multipoint schemes are: a teleconferencing bridge, the IP multicast system, a Multipoint Control Unit (MCU) in a H.320 video conference, etc.

In the data plane, there are two types of data transfer styles: *rooted* and *non-rooted*. In a rooted data plane, a special participant called d_root exists. Data transfer only occurs between the d_root and the rest of the members of this multipoint session, each of which is referred to as a d_leaf. The traffic could be uni-directional, or bi-directional. The data sent out from the d_root will be duplicated (if required) and delivered to every d_leaf, while the data from d_leafs will only go to the d_root. In the case of a rooted data plane, there is no traffic allowed among d_leafs. An example of a protocol that is rooted in the data plane is ST-II.

In a non-rooted data plane, all the participants are equal in the sense that any data they send will be delivered to all the other participants in the same multipoint session. Likewise each d_leaf node will be able to receive data from all other d_leafs, and in some cases, from other nodes which are not participating in the multipoint session as well. No special d_root node exists. IP-multicast is non-rooted in the data plane.

Note that the question of where data unit duplication occurs, or whether a shared single tree or multiple shortest-path trees are used for multipoint distribution are protocol issues, and are irrelevant to the interface the application would use to perform multipoint communications. Therefore these issues are not addressed either in this appendix or by the WinSock interface.

The following table depicts the taxonomy described above and indicates how existing schemes fit into each of the categories. Note that there does not appear to be any existing schemes that employ a non-rooted control plane along with a rooted data plane.

	rooted control plane	**non-rooted (implicit rooted) control plane**
rooted data plane	ATM, ST-II	No known examples.
non-rooted data plane	T.120	IP-multicast, H.320 (MCU)

B.3 WinSock 2 Interface Elements for Multipoint and Multicast

The mechanisms that have been incorporated into WinSock 2 for utilizing multipoint capabilities can be summarized as follows:

- Three attribute bits in the WSAPROTOCOL_INFO struct
- Four flags defined for the *dwFlags* parameter of **WSASocket()**
- One function, **WSAJoinLeaf()**, for adding leaf nodes into a multipoint session
- Two **WSAIoctl()** command codes for controlling multipoint loopback and the scope of multicast transmissions.

The paragraphs which follow describe these interface elements in more detail.

B.3.1. Attributes in WSAPROTOCOL_INFO struct

In support of the taxonomy described above, three attribute fields in the WSAPROTOCOL_INFO structure are use to distinguish the different schemes used in the control and data planes respectively :

(1) XP1_SUPPORT_MULTIPOINT with a value of 1 indicates this protocol entry supports multipoint communications, and that the following two fields are meaningful.
(2) XP1_MULTIPOINT_CONTROL_PLANE indicates whether the control plane is rooted (value = 1) or non-rooted (value = 0).
(3) XP1_MULTIPOINT_DATA_PLANE indicates whether the data plane is rooted (value = 1) or non-rooted (value = 0).

Note that two WSAPROTOCOL_INFO entries would be present if a multipoint protocol supported both rooted and non-rooted data planes, one entry for each.

The application can use **WSAEnumProtocols()** to discover whether multipoint communications is supported for a given protocol and, if so, how it is supported with respect to the control and data planes, respectively.

B.3.2. Flag bits for WSASocket()

In some instances sockets joined to a multipoint session may have some behavioral differences from point-to-point sockets. For example, a d_leaf socket in a rooted data plane scheme can only send information to the d_root participant. This creates a need for the application to be able to indicated the intended use of a socket coincident with its creation. This is done through four flag bits that can be set in the *dwFlags* parameter to **WSASocket()**:

- WSA_FLAG_MULTIPOINT_C_ROOT, for the creation of a socket acting as a c_root, and only allowed if a rooted control plane is indicated in the corresponding WSAPROTOCOL_INFO entry.
- WSA_FLAG_MULTIPOINT_C_LEAF, for the creation of a socket acting as a c_leaf, and only allowed if XP1_SUPPORT_MULTIPOINT is indicated in the corresponding WSAPROTOCOL_INFO entry.
- WSA_FLAG_MULTIPOINT_D_ROOT, for the creation of a socket acting as a d_root, and only allowed if a rooted data plane is indicated in the corresponding WSAPROTOCOL_INFO entry.
- WSA_FLAG_MULTIPOINT_D_LEAF, for the creation of a socket acting as a d_leaf, and only allowed if XP1_SUPPORT_MULTIPOINT is indicated in the corresponding WSAPROTOCOL_INFO entry.

Note that when creating a multipoint socket, exactly one of the two control plane flags, and one of the two data plane flags must be set in **WSASocket()**'s *dwFlags* parameter. Thus, the four possibilities for creating multipoint sockets are: "c_root/d_root", "c_root/d_leaf", "c_leaf/d_root", or "c_leaf /d_leaf".

B.3.3. SIO_MULTIPOINT_LOOP command code for WSAIoctl()

When d_leaf sockets are used in a non-rooted data plane, it is generally desirable to be able to control whether traffic sent out is also received back on the same socket. The SIO_MULTIPOINT_LOOP command code for **WSAIoctl()** is used to enable or disable loopback of multipoint traffic.

B.3.4. SIO_MULTICAST_SCOPE command code for WSAIoctl()

When multicasting is employed, it is usually necessary to specify the scope over which the multicast should occur. Scope is defined as the number of routed network segments to be covered. A scope of zero would indicate that the multicast transmission would not be placed "on the wire" but could be disseminated across sockets within the local host. A scope value of one (the default) indicates that the transmission will be placed on the wire, but will not cross any routers. Higher scope values determine the number of routers that may be crossed. Note that this corresponds to the time-to-live (TTL) parameter in IP multicasting.

B.3.5. WSAJoinLeaf()

The function **WSAJoinLeaf()** is used to join a leaf node into the multipoint session. The function prototype is as follows:

> **SOCKET WSAAPI WSAJoinLeaf (SOCKET** *s*, **const struct sockaddr FAR *** *name*, **int** *namelen*, **LPWSABUF** *lpCallerData*, **LPWSABUF** *lpCalleeData*, **LPQOS** *lpSQOS*, **LPQOS** *lpGQOS*, **DWORD** *dwFlags* **);**

See below for a discussion on how this function is utilized.

B.4. Semantics for joining multipoint leaves

In the following, a multipoint socket is frequently described by defining its role in one of the two planes (control or data). It should be understood that this same socket has a role in the other plane, but this is not mentioned in order to keep the references short. For example when a reference is made to a "c_root socket" , this could be either a c_root/d_root or a c_root/d_leaf socket.

In rooted control plane schemes, new leaf nodes are added to a multipoint session in one or both of two different ways. In the first method, the root uses **WSAJoinLeaf()** to initiate a connection with a leaf node and invite it to become a participant. On the leaf node, the peer application must have created a c_leaf socket and used **listen()** to set it into listen mode. The leaf node will receive an FD_ACCEPT indication when invited to join the session, and signals its willingness to join by calling **WSAAccept()**. The root application will receive an FD_CONNECT indication when the join operation has been completed.

In the second method, the roles are essentially reversed. The root application creates a c_root socket and sets it into listen mode. A leaf node wishing to join the session creates a c_leaf socket and uses **WSAJoinLeaf()** to initiate the connection and request admittance. The root application receives FD_ACCEPT when an incoming admittance request arrives, and admits the leaf node by calling **WSAAccept()**. The leaf node receives FD_CONNECT when it has been admitted.

In a non-rooted control plane, where all nodes are c_leaf's, the **WSAJoinLeaf()** is used to initiate the inclusion of a node into an existing multipoint session. An FD_CONNECT indication is provided when the join has been completed and the returned socket descriptor is useable in the multipoint session. In the case of IP multicast, this would correspond to the IP_ADD_MEMBERSHIP socket option. [4]

There are, therefore, three instances where an application would use **WSAJoinLeaf()**:

1. Acting as a multipoint root and inviting a new leaf to join the session
2. Acting as a leaf making an admittance request to a rooted multipoint session
3. Acting as a leaf seeking admittance to a non-rooted multipoint session (e.g. IP multicast)

B.4.1. Using WSAJoinLeaf()

As mentioned previously, the function **WSAJoinLeaf()** is used to join a leaf node into a multipoint session. **WSAJoinLeaf()** has the same parameters and semantics as **WSAConnect()** except that it returns a socket descriptor (as in **WSAAccept())**, and it has an additional *dwFlags* parameter. The *dwFlags* parameter is used to indicate whether the socket will be acting only as a sender, only as a receiver, or both. Only multipoint sockets may be used for input parameter *s* in this function. If the multipoint socket is in the non-blocking mode, the returned socket descriptor will not be useable until after a corresponding FD_CONNECT indication has been received. A root application in a multipoint session may call **WSAJoinLeaf()** one or more times in order to add a number of leaf nodes, however at most one multipoint connection request may be outstanding at a time.

The socket descriptor returned by **WSAJoinLeaf(**) is different depending on whether the input socket descriptor, *s*, is a c_root or a c_leaf. When used with a c_root socket, the *name* parameter designates a particular leaf node to be added and the returned socket descriptor is a c_leaf socket corresponding to the newly added leaf node. It is not intended to be used for exchange of multipoint data, but rather is used to receive FD_XXX indications (e.g. FD_CLOSE) for the connection that exists to the particular c_leaf. Some multipoint implementations may also allow this socket to be used for "side chats" between the root and an individual leaf node. An FD_CLOSE indication will be received for this socket if the corresponding leaf node calls **closesocket()** to drop out of the multipoint session. Symmetrically, invoking **closesocket()** on the c_leaf socket returned from **WSAJoinLeaf()** will cause the socket in the corresponding leaf node to get FD_CLOSE notification.

When **WSAJoinLeaf()** is invoked with a c_leaf socket, the *name* parameter contains the address of the root application (for a rooted control scheme) or an existing multipoint session (non-rooted control scheme), and the returned socket descriptor is the same as the input socket descriptor. In a rooted control scheme, the root application would put its c_root socket in the listening mode by calling **listen()**. The standard FD_ACCEPT notification will be delivered when the leaf node requests to join itself to the

---

[4] Readers familiar with IP multicast's use of the connectionless UDP protocol may be concerned by the connection-oriented semantics presented here. In particular the notion of using WSAJoinLeaf() on a UDP socket and waiting for an FD_CONNECT indication may be troubling. There is, however, ample precedent for applying connection-oriented semantics to connectionless protocols. It is allowed and sometime useful, for example, to invoke the standard **connect()** function on a UDP socket. The general result of applying connection-oriented semantics to connectionless sockets is a restriction in how such sockets may be used, and such is the case here as well. A UDP socket used in WSAJoinLeaf() will have certain restrictions, and waiting for an FD_CONNECT indication (which in this case simply indicates that the corresponding IGMP message has been sent) is one such limitation.

multipoint session. The root application uses the usual **accept()/WSAAccept()** functions to admit the new leaf node. The value returned from either **accept()** or **WSAAccept()** is also a c_leaf socket descriptor just like those returned from **WSAJoinLeaf()**. To accommodate multipoint schemes that allow both root-initiated and leaf-initiated joins, it is acceptable for a c_root socket that is already in listening mode to be used as in input to **WSAJoinLeaf()**.

A multipoint root application is generally responsible for the orderly dismantling of a multipoint session. Such an application may use **shutdown()** or **closesocket()** on a c_root socket to cause all of the associated c_leaf sockets, including those returned from **WSAJoinLeaf()** and their corresponding c_leaf sockets in the remote leaf nodes, to get FD_CLOSE notification.

B.5. Semantic differences between multipoint sockets and regular sockets

In the control plane, there are some significant semantic differences between a c_root socket and a regular point-to-point socket:

(1) the c_root socket can be used in **WSAJoinLeaf()** to join a new a leaf;
(2) placing a c_root socket into the listening mode (by callings **listen()**) does not preclude the c_root socket from being used in a call to **WSAJoinLeaf()** to add a new leaf, or for sending and receiving multipoint data;
(3) the closing of a c_root socket will cause all the associated c_leaf sockets to get FD_CLOSE notification.

There is no semantic differences between a c_leaf socket and a regular socket in the control plane, except that the c_leaf socket can be used in **WSAJoinLeaf()**, and the use of c_leaf socket in **listen()** indicates that only multipoint connection requests should be accepted.

In the data plane, the semantic differences between the d_root socket and a regular point-to-point socket are

(1) the data sent on the d_root socket will be delivered to all the leaves in the same multipoint session;
(2) the data received on the d_root socket may be from any of the leaves.

The d_leaf socket in the rooted data plane has no semantic difference from the regular socket, however, in the non-rooted data plane, the data sent on the d_leaf socket will go to all the other leaf nodes, and the data received could be from any other leaf nodes. As mentioned earlier, the information about whether the d_leaf socket is in a rooted or non-rooted data plane is contained in the corresponding WSAPROTOCOL_INFO structure for the socket.

B.6. How existing multipoint protocols support these extensions

In this section we indicate how IP multicast and ATM point-to-multipoint capabilities would be accessed via the WinSock 2 multipoint functions. We chose these two as examples because they are very popular and well understood.

B.6.1. IP multicast

IP multicast falls into the category of non-rooted data plane and non-rooted control plane. All applications play a leaf role. Currently most IP multicast implementations use a set of socket options proposed by Steve Deering to the IETF. Five operations are made thus available:

- IP_MULTICAST_TTL - set time to live, controls scope of multicast session
- IP_MULTICAST_IF - determine interface to be used for multicasting
- IP_ADD_MEMBERSHIP - join a specified multicast session
- IP_DROP_MEMBERSHIP - drop out of a multicast session

- IP_MULTICAST_LOOP - control loopback of multicast traffic

Setting the time-to-live for an IP multicast socket maps directly to using the SIO_MULTICAST_SCOPE command code for **WSAIoctl()**. The method for determining the IP interface to be used for multicasting is via a TCP/IP-specific socket option as described in the WinSock 2 Protocol Specific Annex.

The remaining three operations are covered well with the WinSock 2 semantics described here. The application would open sockets with c_leaf/d_leaf flags in **WSASocket()**. It would use **WSAJoinLeaf()** to add itself to a multicast group on the default interface designated for multicast operations. If the flag in **WSAJoinLeaf()** indicates that this socket is only a sender, then the join operation is essentially a no-op and no IGMP messages need to be sent. Otherwise, an IGMP packet is sent out to the router to indicate interests in receiving packets sent to the specified multicast address. Since the application created special c_leaf/d_leaf sockets used only for performing multicast, the standard **closesocket()** function would be used to drop out of the multicast session. The SIO_MULTIPOINT_LOOP command code for **WSAIoctl()** provides a generic control mechanism for determining whether data sent on a d_leaf socket in a non-rooted multipoint scheme will be also received on the same socket.

B.6.2. ATM Point to Multipoint

ATM falls into the category of rooted data and rooted control planes. An application acting as the root would create c_root sockets and counterparts running on leaf nodes would utilize c_leaf sockets. The root application will use **WSAJoinLeaf()** to add new leaf nodes. The corresponding applications on the leaf nodes will have set their c_leaf sockets into listen mode. **WSAJoinLeaf()** with a c_root socket specified will be mapped to the Q.2931 ADDPARTY message. The leaf-initiated join is not supported in ATM UNI 3.1, but will be supported in ATM UNI 4.0. Thus **WSAJoinLeaf()** with a c_leaf socket specified will be mapped to the appropriate ATM UNI 4.0 message.

Appendix C. The Lame List

Keith Moore of Microsoft gets the credit for starting this, but other folks have begun contributing as well. Bob Quinn, from sockets.com, is the kind soul who provided the elaborations on why these things are lame and what to do instead. This is a snapshot of the list as we went to print (plus a few extras thrown in at the last minute).

The Windows Sockets Lame List
(or What's Weak This Week)
---
brought to you by The Windows Sockets Vendor Community

1. Calling connect() on a non-blocking socket, getting WSAEWOULDBLOCK, then immediately calling recv() and expecting WSAEWOULDBLOCK before the connection has been established. *Lame*.

Reason: This assumes that the connection will never be established by the time the application calls recv(). *Lame assumption.*

Alternative: Don't do that. An application using a non-blocking socket must handle the WSAEWOULDBLOCK error value, but must not depend on occurrence of the error.

2. Calling select() with three empty FD_SETs and a valid TIMEOUT structure as a sleazy delay function. *Inexcusably lame*.

Reason: The select() function is intended as a network function, not a general purpose timer.

Alternative: Use a legitimate system timer service.

3. Polling with connect() on a non-blocking socket to determine when the connection has been established. *Dog lame*.

Reason: The WinSock 1.1 spec does not define an error for connect() when a non-blocking connection is pending, so the error value returned may vary.

Alternative: Using asynchronous notification of connection completion is the recommended alternative. An application that prefers synchronous operation mode could use the select() function (but see 23).

Non-Alternative: Changing a non-blocking socket to blocking mode to block on send() or recv() is even more lame than polling on connect().

4. Assuming socket handles are always less than 16. *Mired in a sweaty mass of lameness.*

Reason: The only invalid socket handle value is defined by the

WinSock.H file as INVALID_SOCKET. Any other value the SOCKET type can handle is fair game, and an application *must* handle it. In any case, socket handles are supposed to be opaque, so applications shouldn't depend on specific values for any reason.

Alternative: Expect a socket handle of any value, including 0. And don't expect socket handle values to change with each successive call to socket() or WSASocket(). Socket handles may be reused by the WinSock implementation.

5. Polling with select() and a zero timeout in Win16's non-preemptive environment. *Nauseatingly lame.*

Reason: With any non-zero timeout, select() will call the current blocking hook function, so an application anticipating an event will yield to other processes executing in a 16-bit Windows environment. However, with a zero timeout an application will not yield to other processes, and may not even allow network operations to occur (so it will loop forever).

Alternative: Use a small non-zero timeout. Better yet, use asynchronous notification instead of using select().

6. Calling WSAAsyncSelect() with a zero Event mask just to make the socket non-blocking. *Lame. Lame. Lame. Lame. Lame.*

Reason: WSAAsyncSelect() is designed to allow an application to register for asynchronous notification of network events. The v1.1 WinSock specification didn't specify an error for a zero event mask, but may interpret it as an invalid input argument (so it may fail with WSAEINVAL), or silently ignore the request.

Alternative: To make a socket non-blocking without registering for asynchronous notification, use ioctlsocket() FIONBIO. That's what it's for.

7. Telnet applications that neither enable OOBINLINE, nor read OOB data. *Violently lame.*

Reason: It is not uncommon for Telnet servers to generate urgent data, like when a Telnet client will send a Telnet BREAK command or Interrupt Process command. The server then employs a “Synch” mechanism which consists of a TCP Urgent notification coupled with the Telnet DATA MARK command. If the telnet client doesn’t read the urgent data, then it won’t get any more normal data. Not ever, ever, ever, ever.

Alternative: Every telnet client should be able to read and/or detect OOB data. They should either enable inline OOB data by calling setsockopt() SO_OOBINLINE, or use WSAAsyncSelect() (or WSAEventSelect()) with FD_OOB or select() using exeptfds to detect OOB data arrival, and call recv()/WSARecv() with MSG_OOB

in response.

8. Assuming 0 is an invalid socket handle value. *Uncontrollably lame.*

Reason and Alternative: See item 4.

9. Applications that don't properly shutdown when the user closes the main window while a blocking API is in progress. *Totally lame.*

Reason: WinSock applications that don't close sockets, and call WSACleanup(), may not allow a WinSock implementation to reclaim resources used by the application. Resource leakage can eventually result in resource starvation by all other WinSock applications (i.e. network system failure).

Alternative: While a blocking API is in progress in a 16-bit WinSock 1.1 application, the proper way to abort is to:

1) Call WSACancelBlockingCall()
2) Wait until the pending function returns. If the cancellation occurs before the operation completes, the pending function will fail with the WSAEINTR error, but applications must also be prepared for success, due to the race condition involved with cancellation.
3) Close this socket, and all other sockets. Note: the proper closure of a connected stream socket, involves:
   a) call shutdown() how=1
   b) loop on recv() until it returns 0 or fails with any error
   c) call closesocket()
4) Call WSACleanup()

NOTE: This procedure is *not* relevant to 32-bit WinSock 2 applications, since they really block, so calling WSACancelBlockingCall() from the same thread is impossible.

10. Out of band data. *Intensely lame.*

Reason: Basically TCP can't do Out of Band (OOB) data reliably. If that isn't enough, there are incompatible differences in the implementation at the protocol level (in the urgent pointer offset). Berkeley Software Distribution's (BSD) implementation does RFC 793, literally, and many others implement the corrected RFC 1122 version some versions also allow multiple OOB data bytes by using the start of the MAC frame as the starting point for the offset) If two TCP hosts have different OOB versions, they cannot send OOB data to each other.

Alternative: Ideally, you can use a separate socket for urgent data, although in reality it is inescapable sometimes. Some protocols require it (see item 7), in which case you need to minimize your dependence, or beef up your technical support staff to handle user calls.

11. Calling strlen() on a hostent structure's ip address, then

truncating it to four bytes, thereby overwriting part of malloc()'s heap header. *In all my years of observing lameness, I have seldom seen something this lame.*

Reason: This doesn't really need a reason, does it?

Alternative: Clearly, the only alternative is a brain transplant.

12. Polling with recv(MSG_PEEK) to determine when a complete message has arrived. *Thrashing in a sea of lameness.*

Reason: A stream socket (TCP) does not preserve message boundaries (see item 20). An application that uses recv() MSG_PEEK or ioctlsocket() FIONREAD to wait for a complete message to arrive, may never succeed. One reason might be the internal service provider's buffering; if the bytes in a "message" straddle a system buffer boundary, the WinSock may never report the bytes that exist in other buffers.

Alternative: Don't use peek reads. Always read data into your application buffers, and examine the data there.

13. Passing a longer buffer length than the actual buffer size since you know you won't receive more than the actual buffer size. *Universally lame.*

Reason: WinSock implementations often check buffers for readability or writability before using them to avoid Protection Faults. When a buffer length is longer than the actual buffer length, this check will fail, so the function call will fail (with WSAEFAULT).

Alternative: Always pass a legitimate buffer length.

14. Bounding every set of operations with calls to WSAStartup() and WSACleanup(). *Pushing the lameness envelope.*

Reason: This is not illegal, as long as each WSAStartup() has a matching call to WSACleanup(), but it is more work than necessary.

Alternative: In a DLL, custom control or class library, it is possible to register the calling client based on a unique task handle or process ID. This allows automatic registration without duplication. Automatic de-registration can occur when a process closes its last socket. This is even easier if you use the process notification mechanisms available in the 32-bit environment.

15. Ignoring API errors. *Glaringly lame.*

Reason: Error values are your friends! When a function fails, the error value returned by WSAGetLastError() or included in an asynchronous message can tell you *why* it failed. Based on the function that failed, and the socket state, you can often infer what happened, why, and what to do about it.

Alternative: Check for error values, and write your applications to anticipate them, and handle them gracefully when appropriate.

When a fatal error occurs, always display an error message that shows:
- the function that failed
- the WinSock error number, and/or macro
- a short description of the error meaning
- suggestions for how to remedy, when possible

16. Calling recv() MSG_PEEK in response to an FD_READ async notification message. *Profoundly lame.*

Reason: It's redundant It's redundant.

Alternative: Call recv() in response to an FD_READ message. It may fail with WSAEWOULDBLOCK, but this is easy to ignore, and you are guaranteed to get another FD_READ message later since there is data pending.

17. Installing an empty blocking hook that just returns FALSE. *Floundering in an endless desert of lameness.*

Ed. Note: Fortunately, this is not an issue for WinSock 2 applications, since blocking hooks are now a thing of the past!! (Good Riddance)

Reason: One of the primary purposes of the blocking hook function was to provide a mechanism for an application with a pending blocking operation to yield. By returning FALSE from the blocking hook function, you defeat this purpose and your application will prevent multitasking in the non-preemptive 16-bit Windows environment. This may also prevent some WinSock implementations from completing the pending network operation.

Alternative: Typically this hack is done to try to prevent reentrant messages. There are better ways to do this, like subclassing the active window, although, admittedly, preventing reentrant messages is not an easy problem to avoid.

18. Client applications that bind to a specific port. *Suffocating in self lameness.*

Reason: By definition, client applications actively initiate a network communication, unlike server applications which passively wait for communication. A server must bind() to a specific port which is known to clients that need to use the service, however, a client need not bind() its socket to a specific port in order to communicate with a server.

Not only is it unnecessary for all but a very few application protocols, it is dangerous for a client to bind() to a specific port number. There is a danger in conflicting with another socket that is already using the port number, which would cause the call to bind() to fail with WSAEADDRINUSE.

Alternative: Simply let the WinSock implementation assign the local port number implicitly when you call connect() (on stream

or datagram sockets), or sendto() (on datagram sockets).

19. Nagle challenged applications. *Perilously teetering on the edge of a vast chasm of lameness.*

Reason: The Nagle algorithm reduces trivial network traffic. In a nutshell, the algorithm says don't send a TCP segment until either:
- all outstanding TCP segments have been acknowledged
or
- there's a full TCP segment ready to send

A "Nagle challenged application" is one that cannot wait until either of these conditions occurs, but has such time-critical data that it must send continuously. This results in wasteful network traffic.

Alternative: Don't write applications that depend on the immediate data echo from the remote TCP host.

20. Assuming stream sockets maintain message frame boundaries. *Mind bogglingly lame.*

Reason: Stream sockets (TCP) are called stream sockets, because they provide data streams (duh). As such, the largest message size an application can ever depend on is one-byte in length. No more, no less. This means that with any call to send() or recv(), the WinSock implementation may transfer any number of bytes less than the buffer length specified.

Alternative: Whether you use a blocking or non-blocking socket, on success you should always compare the return from send() or recv() with the value you expected. If it is less than you expected, you need to adjust the buffer length, and pointer, for another function call (which may occur asynchronously, if you are using asynchronous operation mode).

21. 16-bit DLLs that call WSACleanup() from their WEP. *Inconceivably lame.*

Reason: WEP() is lame, ergo depending on it is lame. Seriously, 16-bit Windows did not guarantee that WEP() would always be called, and the Windows subsystem was often in such a hairy state that doing *anything* in WEP() was dangerous.

Alternative: Stay away from WEP().

22. Single byte send()s and recv()s. *Festering in a pool of lameness.*

Reason: Couple one-byte sends with Nagle disabled, and you have at best a 40:1 overhead-to-data ratio. Can you say wasted bandwidth? I thought you could.

As for one-byte receives, think of the effort and inefficiency involved with trying to drink a Guinness Stout through a

hypodermic needle. That's about how your application would feel "drinking" data one-byte at a time.

Alternative: Consider Postel's RFC 793 words to live by: "Be conservative in what you do, be liberal in what you accept from others." In other words, send modest amounts, and receive as much as possible.

23. select(). *Self abusively lame.*

Reason: Consider the steps involved in using select(). You need to use the macros to clear the 3 fdsets, then set the appropriate fdsets for each socket, then set the timer, then call select().

Then after select() returns with the number of sockets that have done something, you need to go through all the fdsets and all the sockets using the macros to find the event that occurred, and even then the (lack of) resolution is such you need to infer the event from the current socket state.

Alternative: Use asynchronous operation mode (e.g. WSAAsyncSelect() or WSAEventSelect()).

24. Applications that call gethostbyname() before calling inet_addr(). *Words fail to express such all consuming lameness.*

Reason: Some users prefer to use network addresses, rather than hostnames at times. The v1.1 WinSock specification does not say what gethostbyname() should do with an IP address in standard ASCII dotted IP notation. As a result, it may succeed and do an (unnecessary) reverse-lookup, or it may fail.

Alternative: With any destination input by a user--which may be a hostname OR dotted IP address--you should call inet_addr() *FIRST* to check for an IP address, and if that fails call gethostbyname() to try to resolve it.

Furthermore, in some applications, you may want to explicitly check the input string for the broadcast address "255.255.255.255," since the return value from inet_addr() for this address is the same as SOCKET_ERROR.

25. Win32 applications that install blocking hooks. *Grossly lame.*

Reason: Besides yielding to other applications (see item 17), blocking hook functions were originally designed to allow concurrent processing within a task while there was a blocking operation pending. In Win32, there's threading.

Alternative: Use threads.

26. Polling with ioctlsocket(FIONREAD) on a stream socket until a complete "message" arrives. *Exceeds the bounds of earthly lameness.*

Reason and Alternative: see item 12

27. Assuming that a UDP datagram of any length may be sent. *Criminally lame.*

Reason: various networks all have their limitations on maximum transmission unit (MTU). As a result, fragmentation will occur, and this increases the likelihood of a corrupted datagram (more pieces to lose or corrupt). Also, the TCP/IP service providers at the receiving end may not be capable of re-assembling a large, fragmented datagram.

Alternative: check for the maximum datagram size with the SO_MAX_MSGSIZE socket option, and don't send anything larger. Better yet, be even more conservative. A max of 8K is a good rule-of-thumb.

28. Assuming the UDP transmissions (especially multicast transmissions) are reliable. *Sinking in a morass of lameness.*

Reason: UDP has no reliability mechanisms (that's why we have TCP).

Alternative: Use TCP and keep track of your own message boundaries.

29. Applications that require vendor-specific extensions, and cannot run (or wore yet, load) without them. *Stooping to unspeakable depths of lameness*

Reason: If you can't figure out the reason, it's time to hang up your keyboard.

Alternative: Have a fallback position that uses only base capabilities for when the extension functions are not present.

30. Expecting errors when UDP datagrams are dropped by the sender, receiver, or any router along the way. *Seeping lameness from every crack and crevice.*

Reason: UDP is unreliable. TCP/IP stacks don't have to tell you when they throw your datagrams away (a sender or receiver may do this when they don't have buffer space available, and a receiver will do it if they cannot reassemble a large fragmented datagram.

Alternative: Expect to lose datagrams, and deal. Implement reliability in your application protocol, if you need it (or use TCP, if your application allows it).

Appendix D. For Further Reference

This specification is intended to cover the Windows Sockets interface in detail. Many details of specific protocols and Windows, however, are intentionally omitted in the interest of brevity, and this specification often assumes background knowledge of these topics. For more information, the following references may be helpful:

D.1 Networking books:

Braden, R.[1989], *RFC 1122, Requirements for Internet Hosts--Communication Layers*, Internet Engineering Task Force.

Comer, D. [1991], *Internetworking with TCP/IP Volume I: Principles, Protocols, and Architecture*, Prentice Hall, Englewood Cliffs, New Jersey.

Comer, D. and Stevens, D. [1991], *Internetworking with TCP/IP Volume II: Design, Implementation, and Internals*, Prentice Hall, Englewood Cliffs, New Jersey.

Comer, D. and Stevens, D. [1991], *Internetworking with TCP/IP Volume III: Client-Server Programming and Applications*, Prentice Hall, Englewood Cliffs, New Jersey.

Leffler, S. et al., *An Advanced 4.3BSD Interprocess Communication Tutorial.*

Stevens, W.R. [1990], *Unix Network Programming*, Prentice Hall, Englewood Cliffs, New Jersey.

Stevens, W.R. [1994]. *TCP/IP Illustrated, Volume 1: The Protocols*, Addison-Wesley, Massachusetts

Wright, G.R. and Stevens, W.R. [1995], *TCP/IP Illustrated Volume 2: The Implementation*, Addison-Wesley., Massachusetts

D.2 Windows Sockets programming books:

Bonner, P. [1995], *Network Programming with Windows Sockets*, ISBN: 0-13-230152-0, Prentice Hall, Englewood Cliffs, New Jersey.

Dumas, A. [1995], *Programming WinSock*, ISBN: 0-672-30594-1, Sams Publishing, Indianapolis, Indiana

Quinn, B. and Shute, D. [1995], *Windows Sockets Network Programming*, ISBN: 0-201-63372-8, Addison-Wesley Publishing Company, Reading, Massachusetts

Roberts, D. [1995], *Developing for the Internet with Winsock*, ISBN 1-883577-42-X, Coriolis Group Books.