Legal Information

Windows Sockets 2 Application Program Interface

Welcome To Windows Sockets 2

This document describes the Windows Sockets 2 Application Programming Interface (API). It consists, primarily, of information from the Windows Sockets 2 API specification, but also includes additional information. The information in this document is not presented in exactly the same way as specification.

Using This Document

This document provides the on-line material needed to create a Windows Sockets application for the Windows NT and the Windows 95 operating systems, using the Microsoft implementation of Windows Sockets 2. It is intended as a reference tool and outlines the functions in the Windows Sockets API.

You should be familiar with Win32 programming concepts to make the best use of this document. Thus, you may want to refer to other references that provide a more systematic guide to writing Windows Sockets applications.

Note This documentation is intended for application developers. If you are developing a transport or service provider, see the "Service Provider Documentation" installed with the Win32 SDK.

Overview of Windows Sockets 2

Windows Sockets 2 utilizes the sockets paradigm that was first popularized by Berkeley Software Distribution (BSD) UNIX. It was later adapted for Microsoft Windows in the Windows Sockets 1.1.

One of the primary goals of Windows Sockets 2 has been to provide a protocol-independent interface fully capable of supporting the emerging networking capabilities, such as real-time multimedia communications.

Windows Sockets 2 is an interface, 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.

Windows Sockets programming previously centered around TCP/IP. Some of the programming practices that worked with TCP/IP do not work with every protocol. As a result, the Windows Sockets 2 API added new functions where necessary.

Windows Sockets 2 has changed its architecture to provide easier access to multiple transport protocols. Following the Windows Open System Architecture (WOSA) model, Windows Sockets 2 now defines a standard service provider interface (SPI) between the application programming interface (API), with its functions exported from WS2_32.DLL, and the protocol stacks. Consequently, Windows Sockets 2 support is not limited to TCP/IP protocol stacks as is the case for Windows Sockets 1.1. For more information, see *Windows Sockets 2 Architecture*.

There are new challenges in developing Windows Sockets 2 applications. When sockets only supported TCP/IP, a developer could create an application that supported only two socket types: connectionless and connection-oriented. Connectionless protocols used SOCK_DGRAM sockets and connection-oriented protocols used SOCK_STREAM sockets. Now, these are just two of the many new socket types. Additionally, developers can no longer rely on socket type to describe all the essential attributes of a transport protocol.

Windows Sockets 2 Features

The new Windows Sockets 2 extends functionality in a number of areas.

Windows Sockets 2 Features

Access to protocols other than TCP/IP	Windows Sockets 2 allows an application to use the familiar socket interface to achieve simultaneous access to a number of installed transport protocols.
Overlapped I/O with scatter/gather	Windows Sockets 2 incorporates the overlapped paradigm for socket I/O and incorporates scatter/gather capabilities as well, following the model established in Win32 environments.
Protocol-independent name resolution facilities:	Windows Sockets 2 includes a standardized set of functions for querying and working with the myriad of name resolution domains that exist today (for example DNS, SAP, and X.500).
Protocol-independent multicast and multipoint:	Windows Sockets 2 applications discover what type of multipoint or multicast capabilities a transport provides and use these facilities in a generic manner.
Quality of service	Window Sockets 2 establishes conventions applications use 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.
Other frequently requested extensions	Windows Sockets 2 incorporates shared sockets and conditional acceptance; exchange of user data at connection setup/teardown time; and protocol-specific extension mechanisms.

Conventions for New Functions

Windows Sockets 2, with its expanded scope, takes the socket paradigm beyond the original design. As a result, a number of new functions have been added. These have been assigned names that begin with "WSA." In all but a few instances, these new functions are expanded versions of existing functions from BSD sockets.

The new functions are described in the reference section of the document, following the conventions of the Win32 SDK. The new functions are also listed in *Summary of New Functions*.

Microsoft Extensions and Windows Sockets 2

The Windows Sockets 2 specification defines an extension mechanism that exposes advanced transport functionality to application programs. For more information, see Function Extension Mechanism.

The following Microsoft-specific extensions were added to Windows Sockets 1.1. They are also available in Windows Sockets 2.

AcceptEx

GetAcceptExSockaddrs

TransmitFile

WSARecvEx

These functions are not exported from the WS2_32.DLL; they are exported from MSWSOCK.DLL.

An application written to use the Microsoft-specific extensions to Windows Sockets will not run correctly over a Windows Sockets service provider that does not support those extensions.

Socket Handles for Windows Sockets 2

A socket handle can optionally be a file handle In Windows Sockets 2. It is possible to use socket handles with **ReadFile**, **WriteFile**, **ReadFileEx**, **WriteFileEx**, **DuplicateHandle**, and other Win32 functions. Not all transport service providers will support this option. For an application to run over the widest possible number of service providers, it should not assume that socket handles are file handles.

Windows Sockets 2 has expanded certain functions used for transferring data between sockets using handles. The functions offer advantages specific to sockets for transferring data and include **WSARecv**, **WSASend**, and **WSADuplicateSocket**.

New Concepts, Additions and Changes for Windows Sockets 2

This section summarizes Windows Sockets 2 and describes the major changes and additions it contains. Windows Sockets 2 differs from Windows Sockets 1.1 in several ways, particularly in the architecture. The new architecture, discussed in **Windows Sockets 2 Architecture**, provides the foundation for many of the new concepts that have been incorporated into Windows Sockets 2.

An overview of the additions and changes in Windows Sockets 2 follows the discussion of the new architecture.

Many of the functions in Windows Sockets 2 are the same as in the other versions of sockets. However, there are several new functions, which are summarized in **Summary of New Functions**. For detailed information on how to use a specific function or feature, refer to the Reference section.

Windows Sockets 2 Architecture

A number of Windows Sockets 2 features required a substantial change in the Windows Sockets architecture. The resulting architecture is considerably different from previous versions, but the benefits are numerous. Foremost among these is Simultaneous Access to Multiple Transport Protocols, explained in detail in the following section.

Other features include the adoption of protocol-independent name resolution facilities, provisions for layered protocols and protocol chains, and a different mechanism for Windows Sockets service providers to offer extended, provider-specific functionality.

Simultaneous Access to Multiple Transport Protocols

In order to provide simultaneous access to multiple transport protocols, the architecture has changed for Windows Sockets 2. With Windows Sockets 1.1, the DLL that implements the Windows Sockets interface is supplied by the vendor of the TCP/IP protocol stack. The interface between the Windows Sockets DLL and the underlying stack was both unique and proprietary. Windows Sockets 2 changes the model by defining a standard service provider interface (SPI) between the Windows Sockets DLL and protocol stacks. In this way, multiple stacks from different vendors can be accessed simultaneously from a single Windows Sockets DLL. Furthermore, Windows Sockets 2 support is not limited to TCP/IP protocol stacks as it is in Windows Sockets 1.1.

The Windows Open System Architecture (WOSA) compliant Windows Sockets 2 architecture is illustrated as follows:

{ewc msdncd, EWGraphic, bsd23510 0 /a "SDK_1.WMF"}

Windows Sockets 2 Architecture

With the Windows Sockets 2 architecture, it is not necessary, or desirable, for stack vendors to supply their own implementation of WS2_32.DLL, since a single WS2_32.DLL must work across all stacks. The WS2_32.DLL and compatibility shims should be viewed in the same way as an operating system component.

Backward Compatibility For Windows Sockets 1.1 Applications

Windows Sockets 2 has been made backward compatible with Windows Sockets 1.1 on two levels: source and binary. This maximizes interoperability between Windows Sockets applications of any version and Windows Sockets implementations of any version. It also minimizes problems for users of Windows Sockets applications, network stacks, and service providers. Current Windows Sockets 1.1-compliant applications will run over a Windows Sockets 2 implementation without modification of any kind, as long as at least one TCP/IP service provider is properly installed.

Source Code Compatibility

Source code compatibility in Windows Sockets 2 means, with few exceptions, that all the Windows Sockets 1.1 functions are preserved in Windows Sockets 2. Windows Sockets 1.1 applications that make use of blocking hooks will need to be modified since blocking hooks are no longer supported in Windows Sockets 2. (For more information, see Windows Sockets 1.1 Blocking routines & EINPROGRESS.)

Thus, existing Windows Sockets 1.1 application source code can easily be moved to the Windows Sockets 2 system by including the new header file, WINSOCK2.H, and performing a straightforward relink with the appropriate Windows Sockets 2 libraries. Application developers are encouraged to view this as the first step in a full transition to Windows Sockets 2 because there are numerous ways in which a Windows Sockets 1.1 application can be improved by exploring and using the new functionality in Windows Sockets 2.

Binary Compatibility

A major design goal for Windows Sockets 2 was to enable existing Windows Sockets 1.1 applications to work, unchanged at a binary level, with Windows Sockets 2. Since Windows Sockets 1.1 applications are TCP/IP-based, binary compatibility implies that TCP/IP-based Windows Sockets 2 Transport and Name Resolution Service Providers are present in the Windows Sockets 2 system. In order to enable Windows Sockets 1.1 applications in this scenario, the Windows Sockets 2 system has an additional "shim" component supplied with it: a Version 1.1-compliant WINSOCK.DLL.

Installation guidelines for Windows Sockets 2 ensure there will be no negative impact to existing Windows Sockets-based applications on an end user system by the introduction of any Windows Sockets 2 components.

{ewc msdncd, EWGraphic, bsd23510 1 /a "SDK_2A.WMF"}

Windows Sockets 1.1 Compatibility Architecture

Important To obtain information about the underlying TCP/IP stack, Windows Sockets 1.1 applications currently use certain members of the **WSAData** structure (obtained through a call to **WSAStartup**). These members include: *iMaxSockets, iMaxUdpDg,* and *lpVendorInfo*.

While Windows Sockets 2 applications ignore these values (since they cannot uniformly apply to all available protocol stacks), safe values are supplied to avoid breaking Windows Sockets 1.1 applications.

Making Transport Protocols Available To Windows Sockets

A transport protocol must be properly installed on the system and registered with Windows Sockets to be accessible to an application. The WS2_32.DLL exports a set of functions to facilitate the registration process. This includes creating a new registration and removing an existing one.

When new registrations are created, the caller (that is, the stack vendor's installation script) supplies one or more filled in WSAPROTOCOL_INFO structures containing a complete set of information about the protocol. (See the SPI document, installed with the SPK, for information on how this is accomplished.) Any transport stack that is installed this way will be referred to as a Windows Sockets service provider.

The Windows Sockets 2 SDK includes a small Windows applet that allows the user to view and modify the order in which service providers are enumerated. By using this applet, a user can manually establish a particular TCP/IP protocol stack as the default TCP/IP provider if more than one such stack is present.

Layered Protocols and Protocol Chains

Windows Sockets 2 incorporates the concept 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 this type of layered protocol is a security layer that adds a protocol to the socket connection process in order to perform authentication and establish an encryption scheme. Such a security protocol generally requires 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, that is fully capable of performing data communications with a remote endpoint. A *layered protocol* is a protocol that cannot stand alone, while a *protocol chain* is one or more layered protocols strung together and anchored by a base protocol.

A protocol chain is created by having the layered protocols support the Windows Sockets 2 SPI at both their upper and lower edges. A special WSAPROTOCOL_INFO structure is created that refers to the protocol chain as a whole, and that describes the explicit order in which the layered protocols are joined. This is illustrated in the figure *Layered Protocol Architecture*. Since only base protocols and protocol chains are directly usable by applications, they are the only ones listed when the installed protocols are enumerated with the **WSAEnumProtocols** function.

{ewc msdncd, EWGraphic, bsd23510 2 /a "SDK_3.WMF"}

Layered Protocol Architecture

Using Multiple Protocols

An application uses the **WSAEnumProtocols** function to determine which transport protocols and protocol chains are present, and to obtain information about each as contained in the associated WSAPROTOCOL_INFO structure.

In most instances, there is a single WSAPROTOCOL_INFO structure for each protocol or protocol chain. However, some protocols exhibit multiple behaviors. For example, the SPX protocol is message oriented (that is, the sender's message boundaries are preserved by the network), but the receiving socket can ignore these message boundaries and treat it as a byte stream. Thus, two different WSAPROTOCOL_INFO structure entries could exist for SPX—one for each behavior.

In Windows Sockets 2, several new address family, socket type, and protocol values appear. Windows Sockets 1.1 supported a single address family (AF_INET) comprising a small number of well-known socket types and protocol identifiers. The existing address family, socket type, and protocol identifiers are retained for compatibility reasons, but new transport protocols with new media types are supported.

A Windows Sockets 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. This is the World Wide Web URL for the Windows Sockets 2 Identifier Clearinghouse:

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

New, unique identifiers are not necessarily well known, but this should not pose a problem. Applications that need to be protocol-independent are encouraged to select a protocol on the basis of its suitability rather than the values assigned to their *socket_type* or *protocol* fields. Protocol suitability is indicated by the communications attributes, such as message versus byte stream, and reliable versus unreliable, that are contained in the protocol WSAPROTOCOL_INFO structure. Selecting protocols on the basis of suitability as opposed to well-known protocol names and socket types lets protocol-independent applications take advantage of new transport protocols and their associated media types, as they become available.

The server half of a client/server application benefits by establishing listening sockets on all suitable transport protocols. Then, the client can establish its connection using any suitable protocol. For example, this would let a client application be unmodified whether it was running on a desktop system connected through LAN or on a laptop using a wireless network.

Multiple Provider Restrictions on select

The **select** function is used to determine the status of one or more sockets in a set. For each socket, the caller can request information on read, write, or error status. A set of sockets is indicated by an FD_SET structure.

Windows Sockets 2 allows an application to use more than one service provider, but the **select** function is limited to a set of sockets associated with a single service provider. This does not in any way restrict an application from having multiple sockets open through multiple providers.

There are two ways to determine the status of set of sockets that span more than one service provider: 1) using the **WSAWaitForMultipleEvents** or **WSAEventSelect** functions when blocking semantics are employed, and 2) using the **WSAAsyncSelect** function when nonblocking operations are employed.

When an application needs to use blocking semantics on a set of sockets that spans multiple providers, **WSAWaitForMultipleEvents** is recommended. The application can also use the **WSAEventSelect** function, which allows the FD_XXX network events (see **WSAEventSelect**) to associate with an event object and be handled from within the event object paradigm (described in *Overlapped I/O and Event Objects*).

The **WSAAsyncSelect** function is recommended when nonblocking operations are preferred. This function is not restricted to a single provider because it takes a socket descriptor as an input parameter.

Function Extension Mechanism

The Windows Sockets DLL, WS2_32.DLL, is no longer supplied by each individual stack vendor. As a result, it is no longer possible for a stack vendor to offer extended functionality by just adding entry points to the WS2_32.DLL. To overcome this limitation, Windows Sockets 2 takes advantage of the new **WSAIoctl** function to accommodate service providers who want to offer provider-specific functionality extensions. This mechanism assumes, 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 through 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 while the output buffer contains the function pointer itself. The application can then invoke the extension function directly without passing through the WS2_32.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 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.

Debug and Trace Facilities

When the developer of a Windows Sockets 2 application encounters a Windows Sockets-related bug there is a need to isolate the bug in 1) the application, 2) the WS2_32.DLL, or 3) the service provider. Windows Sockets 2 addresses this need through a specially devised version of the WS2_32.DLL and a separate debug/trace DLL. This combination allows all procedure calls across the Windows Sockets 2 API or SPI to be monitored and, to some extent, be 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 be prevented or redirected. With access to this level of information and control, a developer can isolate any problem in the application, WS2_32.DLL, or service provider.

The Windows Sockets 2 SDK includes the debug WS2_32.DLL, a sample debug/trace DLL, and a document containing a detailed description of the 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 specific needs.

Name Resolution

Windows Sockets 2 includes provisions for standardizing the way applications access and use the various network name resolution services. Windows Sockets 2 applications do not need to be aware of the widely differing interfaces associated with name services such as DNS, NIS, X.500, SAP, and others. An introduction to this topic and the details of the functions are currently located in *Protocol-Independent Name Resolution*.

Overlapped I/O and Event Objects

Windows Sockets 2 introduces overlapped I/O and requires that all transport providers support this capability. Overlapped I/O follows the model established in Win32 and can be performed only on sockets that were created through the **WSASocket** function with the WSA_FLAG_OVERLAPPED flag set.

> **Note** Creating a socket with the overlapped attribute has no impact on whether a socket is currently in the blocking or nonblocking mode. Sockets created with the overlapped attribute can be used to perform overlapped I/O—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 the **WSARecv** or **WSARecvFrom** functions 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, that data could be placed in the user's buffers immediately as it arrives. Thus, it can 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. That is, the incoming data is buffered internally until the application issues a receive call and thereby supplies a buffer into which the data can be copied. An exception to this is when the appliation uses **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 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 was completed immediately and that the corresponding completion indication 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 alertable 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 sockets that are byte-stream style, the completion indication occurs whenever the incoming data is exhausted, regardless of whether the buffers are full. 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 can be invoked several times to post receive buffers in preparation for incoming data, and the send functions can be invoked several times to queue multiple buffers to send. While the application can rely upon a series of overlapped send buffers being sent in the order supplied, the corresponding completion indications might occur in a different order. Likewise, on the receiving side, buffers will be filled in the order they are supplied, but the completion indications might occur in a different order.

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

Event Objects

Introducing overlapped I/O requires a mechanism for applications to unambiguously associate send and receive requests with their subsequent completion indications. In Windows Sockets 2, this is accomplished with event objects that are modeled after Win32 events. Windows Sockets event objects are fairly simple constructs that can be created and closed, set and cleared, and waited upon and polled. Their prime utility is 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 that can 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 *Asynchronous Notification Using Event Objects*.

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

Receiving Completion Indications

Several options are available for receiving completion indications, thus providing applications with appropriate levels of flexibility. These include: waiting (or blocking) on event objects, polling event objects, and socket I/O completion routines.

Blocking and Waiting for Completion Indication

Applications can block while waiting for one or more event objects to become set using the **WSAWaitForMultipleEvents** function. In Win32 implementations, the process or thread will truly block. Since Windows Sockets 2 event objects are implemented as Win32 events, the native Win32 function, **WaitForMultipleObjects** can also be used for this purpose. This is especially useful if the thread needs to wait on both socket and nonsocket events.

Polling for Completion Indication

Applications that prefer not to block can use the **WSAGetOverlappedResult** function 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 the **WSAGetLastError** function to retrieve the error status of the overlapped operation.

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-specific function that will be called after a successfully initiated overlapped I/O operation was completed (successfully or otherwise). The completion routine follows the same rules as stipulated for Win32 file I/O completion routines. That is, the completion routine will not be invoked until the thread is in an alertable wait state, such as when the function **WSAWaitForMultipleEvents** is invoked with the *fAlertable* flag set.

The 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.

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 the application supplies a pointer to a completion function, whether a WSAOVERLAPPED structure is referenced, and by the value of the hEvent member within the WSAOVERLAPPED structure (if supplied). The following table summarizes the completion semantics for an overlapped socket and shows the various combinations of *lpOverlapped*, *hEvent*, and *lpCompletionRoutine:*

lpOverlapped	***hEvent***	***lpCompletionRoutine***	**Completion Indication**
NULL	not applicable	ignored	Operation completes synchronously. It behaves as if it were a nonoverlapped socket.
!NULL	NULL	NULL	Operation completes overlapped, but there is no Windows Sockets 2-supported completion mechanism. The completion port mechanism (if supported) can 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.

Asynchronous Notification Using Event Objects

The **WSAEventSelect** and **WSAEnumNetworkEvents** functions are provided to accommodate applications such as daemons and services that have no user interface (and hence do not use Windows handles). The **WSAEventSelect** function behaves exactly like the **WSAAsyncSelect** function. However, instead of causing a Windows message to be sent on the occurrence of an FD_XXX network event (for example, FD_READ and FD_WRITE), 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 can 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 automatically cleared. If needed, the application can also designate a particular event object that is cleared along with the network event memory.

Quality of Service

The basic Quality of Service (QOS) mechanism in Windows Sockets 2 descends from the flow specification as described in RFC 1363, dated September 1992. Following is a brief overview of this concept:

Flow specifications describe a set of characteristics about a proposed, unidirectional flow through the network. An application can associate a pair of flow specifications 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. Flow specifications indicate parametrically what level of service is required and provide a feedback mechanism for applications to use in adapting to network conditions.

This is the usage model for QOS in Windows Sockets 2. An application can establish its QOS requirements at any time with **WSAIoctl** or coincident with the connect operation with **WSAConnect**. For connection-oriented transports, it is often most convenient for an application to use the **WSAConnect** function because any QOS values supplied at connect time supersede those supplied earlier with the **WSAIoctl** function. If the **WSAConnect** function completes successfully, the application knows that its QOS request has been honored by the network. 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 can scale down its service request and try again, or it can give up.

Transport providers update the associated flowspec structures after every connection attempt (successful or otherwise) in order to indicate, as well as possible, the existing network conditions. (Updating with the **Default Values** will indicate that information about the current network conditions is not available.) This update from the service provider about current network conditions is especially useful when the application's QOS request consists entirely of the default (unspecified) values, which any service provider should be able to meet.

Applications expect to use this information about current network conditions to guide their use of the network, including any subsequent QOS requests. However, the information provided by the transport in the updated flowspec structure is only an indication. It might be little more than a rough estimate that only applies to the first hop and not to the complete, end-to-end connection. The application must take appropriate precautions in interpreting this information.

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

Even after a flow is established, conditions in the network can change or one of the communicating parties might invoke a QOS renegotiation that results in a reduction (or increase) in the available service level. A notification mechanism is included that utilizes the usual Windows Sockets notification techniques (FD_QOS and FD_GROUP_QOS events) to indicate to the application that QOS levels have changed.

A service provider generates FD_QOS/FD_GROUP_QOS notifications when the current level of service supported is significantly different (especially in the negative direction) from what was last reported as a basic guideline. The application should use the **WSAIoctl** function with SIO_GET_QOS and/or SIO_GET_GROUP_QOS to retrieve the corresponding flowspec structure and examine them in order to discover what aspect of the service level has changed. The QOS structures will be updated where appropriate. regardless of whether FD_QOS/FD_GROUP_QOS is registered and generated.

If the updated level of service is not acceptable, the application can 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 specifications proposed for Windows Sockets 2 divide QOS characteristics into the following general areas:

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. Though the bandwidth requirement is expressed in terms of a token rate, a service provider need not implement token buckets. Any traffic management scheme that yields equivalent behavior is permitted.

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

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

Cost
: This is a placeholder for a future time when a meaningful cost metric can be determined.

Provider-specific parameters
: The flow specification itself can be extended in ways that are particular to specific providers.

QOS Templates

It is possible for QOS templates to be established for well-known media flows such as H.323, G.711, and others. 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.

Default Values

A default flowspec structure is associated with each eligible socket at the time it is created. The member values for the default flowspec structure, in all cases, indicate that no particular flow characteristics are being requested from the network. Applications only need to modify values important to that application, but must be aware that there is some coupling between fields such as TokenRate and TokenBucketSize. These are the values for the default flow spec:

```
TokenRate =           -1 (not specified)
TokenBucketSize =     -1 (not specified)
PeakBandwidth =       -1 (not specified)
Latency =             -1 (not specified)
DelayVariation =      -1 (not specified)
LevelOfGuarantee =    BestEffortService
CostOfCall =          0 (reserved for future use)
NetworkAvailability = (read-only value)
```

Socket Groups

Windows Sockets 2 introduces the concept 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's quality of service specification.

Applications needing to exchange multimedia streams over the network benefit by establishing a specific relationship among the set of sockets being utilized. This can include, as a minimum, an indication to the service provider about the relative priorities of the media streams being carried. For example, a conferencing application would likely give the socket used for carrying the audio stream a higher priority than the socket used for the video stream. Furthermore, there are transport providers (for example, digital telephony and ATM) that 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, service providers can operate with maximum effectiveness.

The **WSASocket** and **WSAAccept** functions are two new functions used to specifically create and join a socket group coincident with creating a new socket. Socket group identifiers 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.

Shared Sockets

The **WSADuplicateSocket** function is introduced to enable socket sharing across 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. 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. Sockets can be shared among threads in a given process without using the **WSADuplicateSocket** function because a socket descriptor is valid in all threads of a process.

The two (or more) descriptors that reference a shared socket can be used independently as far as I/O is concerned. However, the Windows Sockets interface does not implement any type of access control, so the processes must coordinate any operations on a shared socket. A typical example of sharing sockets is to use one process for creating sockets and establishing connections. This process then hands off sockets to other processes that are responsible for information exchange.

The **WSADuplicateSocket** function creates socket descriptors and not the underlying socket. As a result, all the states associated with a socket are 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 can 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 the **WSAAsyncSelect** and **WSAEventSelect** functions. 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.

Enhanced Functionality During Connection Setup and Teardown

The **WSAAccept** function lets an application obtain caller information such as caller ID and QOS before deciding whether to accept an incoming connection request. This is done with a callback to an application-supplied condition function.

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

It is also possible (for protocols that support this) to exchange user data between the endpoints at connection teardown time. The end that initiates the teardown can call the **WSASendDisconnect** function to indicate that no more data be sent and to initiate the connection teardown sequence. 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 by the FD_CLOSE indication), the **WSARecvDisconnect** function can be called to receive the disconnect data, if any.

To illustrate how disconnect data can 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 (using disconnect data) the total number of transactions it processed with the server. The server in turn responds with the cumulative 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	(5a) Invoke closesocket
(7) Invoke **WSARecvDisconnect** to receive and store cumulative grand total of transactions	
(8) Invoke **closesocket**	

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

Extended Byte Order Conversion Routines

Windows Sockets 2 does not assume that the network byte order for all protocols is the same. A set of conversion routines is supplied for converting 16-bit and 32-bit quantities to and from network byte order. These routines take as an input parameter an integer whose value is generally a manifest constant that specifies what the desired network byte order is (currently "big-endian" or "little-endian"). Also, the WSAPROTOCOL_INFO structure for each protocol includes a field for use as the input parameter for the byte-ordering functions.

Support for Scatter/Gather I/O

The **WSASend**, **WSASendTo**, **WSARecv**, and **WSARecvFrom** functions all take an array of application buffers as input parameters and can 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 the supplied buffers are utilized.

Protocol-Independent Multicast and Multipoint

Windows Sockets 2 provides a generic method for utilizing the multipoint and multicast capabilities of transports. This generic method implements these features just as it allows the basic data transport capabilities of numerous transport protocols to be accessed. The term multipoint is used hereafter to refer to both multicast and multipoint communications.

Current multipoint implementations (for example, IP multicast, ST-II, T.120, and ATM UNI) vary widely. 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 differ among implementations. The WSAPROTOCOL_INFO structure for Windows Sockets 2 is used to declare the various multipoint attributes of a protocol. By examining these attributes, the programmer knows what conventions to follow with the applicable Windows Sockets 2 functions to setup, utilize and teardown multipoint sessions.

Following is a summary of the features of Windows Sockets 2 that support multipoint:

- Two attribute bits in the WSAPROTOCOL_INFO structure.
- Four flags defined for the *dwFlags* parameter of the **WSASocket**function.
- 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 The inclusion of these multipoint features in Windows Sockets 2 does not preclude an application from using an existing protocol-dependent interface, such as the Deering socket options for IP multicast.

See Multipoint and Multicast Semantics for detailed information on how the various multipoint schemes are characterized and how the applicable features of Windows Sockets 2 are utilized.

Summary of New Socket Options

The new socket options for Windows Sockets 2 are summarized in the following table. See **getsockopt** and **setsockopt** for detailed information on these options. The other new protocol-specific socket options can be found in the Protocol-specific Annex (a separate document included with the Win32 SDK).

Value	Type	Meaning	Default	Note
SO_GROUP_ID	GROUP	The identifier of the group to which this socket belongs.	NULL	get only
SO_GROUP_PRIORITY	int	The relative priority for sockets that are part of a socket group.	0	
SO_MAX_MSG_SIZE	int	Maximum size of a message for message-oriented socket types. 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	

Summary of New Socket Ioctl Opcodes

The new socket ioctl opcodes for Windows Sockets 2 are summarized in the following table. See **WSAIoctl** for detailed information on these opcodes. The **WSAIoctl** function also supports all the ioctl opcodes specified in **ioctlsocket**. Theother new protocol-specific ioctl opcodes can be found in the Protocol-specific Annex (a separate document included with the Win32 SDK).

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 specification(s) for the socket.
SIO_GET_GROUP_QOS	<not used>	QOS	Retrieve current group flow specification(s) for the group this socket belongs to.
SIO_MULTIPOINT_LOOKBACK	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 specification(s) for the socket.
SIO_SET_GROUP_QOS	QOS	<not used>	Establish new group flow specification(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.

Summary of New Functions

The new API functions for Windows Sockets 2 are summarized in the following table.

Data Transport Functions

Function	Description
WSAAccept[1]	An extended version of **accept** which allows for conditional acceptance and socket grouping.
WSACloseEvent	Destroys an event object.
WSAConnect[1]	An extended version of **connect** which allows for exchange of connect data and QOS specification.
WSACreateEvent	Creates an event object.
WSADuplicateSocket	Creates a new socket descriptor for a shared socket.
WSAEnumNetworkEvents	Discovers occurrences of network events.
WSAEnumProtocols	Retrieves information about each available protocol.
WSAEventSelect	Associates network events with an event object.
WSAGetOverlappedResult	Gets completion status of overlapped operation.
WSAGetQOSByName	Supplies QOS parameters based on a well-known service name.
WSAHtonl	Extended version of **htonl**.
WSAHtons	Extended version of **htons**.
WSAIoctl[1]	Overlapped-capable version of **ioctlsocket**.
WSAJoinLeaf[1]	Joins a leaf node into a multipoint session.
WSANtohl	Extended version of **ntohl**.
WSANtohs	Extended version of **ntohs**.
WSARecv[1]	An extended version of **recv** which accommodates scatter/gather I/O, overlapped sockets, and provides the *flags* parameter as IN OUT.
WSARecvDisconnect	Terminates reception on a socket and retrieves the disconnect data, if the socket is connection-oriented.
WSARecvFrom[1]	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[1]	An extended version of **send** which accommodates scatter/gather I/O and overlapped sockets.
WSASendDisconnect	Initiates termination of a socket

	connection and optionally sends disconnect data.
WSASendTo[1]	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 structure as input and allows overlapped sockets to be created. Also allows socket groups to be formed.
WSAWaitForMultipleEvents[1]	Blocks on multiple event objects.

1 The routine can block if acting on a blocking socket.

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 Windows Sockets 1.1 applications to Windows Sockets 2.

Deviation from Berkeley Sockets

There are a few limited instances where Windows Sockets has had to divert from strict adherence to the Berkeley conventions, usually because of difficulties of implementation in a Windows environment.

Socket Data Type

A new data type, SOCKET, has been defined. This is needed because a Windows Sockets 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. Windows Sockets 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 the **socket** and **accept** routines 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)
    {...}
```

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 Windows Sockets. 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.

Error codes - errno, h_errno & WSAGetLastError

Error codes set by Windows Sockets 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. This function is provided in Windows Sockets as a precursor (and eventually an alias) for the Win32 function **GetLastError**. This 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 Windows Sockets 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)
    {...}
```

This specification defines a recommended set of error codes, and lists the possible errors that can be returned as a result of each function. It may be the case in some implementations that other Windows Sockets 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 Windows Sockets will not return any value that is not enumerated in the table of legal Windows Sockets errors given in the section Error Codes.

Pointers

All pointers used by applications with Windows Sockets should be FAR. To facilitate this, data type definitions such as LPHOSTENT are provided.

Renamed functions

In two cases it was necessary to rename functions which are used in Berkeley Sockets in order to avoid clashes with other Win32 API functions.

close & closesocket

Sockets are represented by standard file descriptors in Berkeley Sockets, so the **close** function can be used to close sockets as well as regular files. While nothing in the Windows Sockets prevents an implementation from using regular file handles to identify sockets, nothing requires it either. 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.

ioctl & ioctlsocket/WSAIoctl

Various C language run-time systems use the **ioctl** routine for purposes unrelated to Windows Sockets. As a consequence, the **ioctlsocket** function and the **WSAIoctl** function were defined to handle socket functions that were performed by **ioctl** and **fcntl** in the Berkeley Software Distribution.

Maximum number of sockets supported

The maximum number of sockets supported by a particular Windows Sockets service provider is implementation specific. An application should make no assumptions about the availability of a certain number of sockets. For more information on this topic see **WSAStartup**.

The maximum number of sockets that an application can actually use is independent of the number of sockets supported by a particular implementation. The maximum number of sockets that a Windows Sockets application can use is determined at compile time by the manifest constant FD_SETSIZE. This value is used in constructing the fd_set structures used in **select**. 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, you could add "-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 Windows Sockets service provider.

Include files

A number of standard Berkeley include files are supported for ease of porting existing source code based on Berkeley sockets. However, these Berkeley header files merely include the WINSOCK2.H include file, and it is therefore sufficient (and recommended) that Windows Sockets application source files just include WINSOCK2.H.

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)
    {...}
```

Raw Sockets

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

Byte Ordering

Care must always be taken to account for any differences between the byte ordering used by Intel Architecture and the byte ordering used on the wire by individual transport protocols. Any reference to an address or port number passed to or from a Windows Sockets 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 sockaddr_in structure (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.

Consider an application that normally contacts a server on the TCP port corresponding to the "time" service, but provides a mechanism for the user to specify an alternative port. 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 were to display the number of the port within an address (returned by **getpeername** for example), the port number must be converted from network to host order (using **WSANtohs**) before it can be displayed.

Since the Intel 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 Windows Sockets rather than writing their own conversion code since future implementations of Windows Sockets are likely to run on systems for which the host order is identical to the network byte order. Only applications that use the standard conversion functions are likely to be portable.

Windows Sockets Compatibility Issues

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

- WSACancelBlockingCall
- WSAIsBlocking
- WSASetBlockingHook
- WSAUnhookBlockingHook

Windows Sockets 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 continue to support the complete set of Windows Sockets 1.1 functions. In order for programs to become Windows Sockets 2 applications, some amount of code modification must occur. In most cases, you will substitute the judicious use of threads to accommodate processing that was being accomplished with a blocking hook function.

Default state for a socket's overlapped attribute

The **socket** function created sockets with the overlapped attribute set by default in the first WSOCK32.DLL, the 32-bit version of Windows Sockets 1.1. In order to insure backward 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 with the **socket** function will have the overlapped attribute. However, in order to be more compatible with the rest of the Win32 API, sockets created with **WSASocket** will, by default, NOT have the overlapped attribute. This attribute will only be applied if the WSA_FLAG_OVERLAPPED bit is set.

Windows Sockets 1.1 Blocking routines & EINPROGRESS

One major issue in porting applications from a Berkeley sockets environment to a Windows environment involves "blocking"; that is, invoking a function that does not return until the associated operation is completed. A problem arises when the operation takes an arbitrarily long time to complete: an example is a **recv**, which might 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 nonblocking. Windows Sockets 1.1 environments could not assume pre-emptive scheduling. Therefore, 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.

Even on a blocking socket, some functions – **bind**, **getsockopt**, and **getpeername** for example – complete immediately. There is no difference between a blocking and a nonblocking operation for those functions. Other operations, such as **recv,** can complete immediately or could 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. All routines that can block are listed with an asterisk in the tables above and below.

With Windows Sockets 1.1, a blocking operation that cannot complete immediately is handled by psuedo-blocking as follows. The service provider initiates the operation, 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 Windows Sockets function. If the function has completed, or if **WSACancelBlockingCall** has been invoked, the blocking function completes with an appropriate result.

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

In a Windows Sockets 1.1 environment, 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 Windows Sockets call. Because of the difficulty in managing this condition safely, Windows Sockets 1.1 does not support such application behavior. An application it not permitted to make more than one nested Windows Sockets function calls. Only one outstanding function call will be allowed for a particular task. The only exceptions are two functions that are provided to assist the programmer in this situation: **WSAIsBlocking** and **WSACancelBlockingCall**.

The **WSAIsBlocking** function can be called at any time to determine whether or not a blocking Windows Sockets 1.1 call is in progress. Similarly, the **WSACancelBlockingCall** fucntion can be called at any time to cancel an in-progress blocking call. Any other nesting of Windows Sockets functions will fail with the error WSAEINPROGRESS. It should be emphasized that this restriction applies to both blocking and non-blocking operations.

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 Windows Sockets API includes the function **WSASetBlockingHook**, which allows the application to specify a special routine which will be called instead of the default message dispatch routine described above.

The Windows Sockets provider calls the blocking hook only if all of the following are true: the routine is one that 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 or the **WSAAsyncSelect** function set a socket to nonblocking mode.)

The blocking hook will never be called and the application does not need to be concerned with the re-entrancy issues the blocking hook can introduce if an application follows these guideline:

- It uses only nonblocking sockets, and;
- It uses the **WSAAsyncSelect** and/or the **WSAAsyncGetXByY** routines instead of **select** and the **getXbyY** routines.

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

Graceful shutdown, linger options and socket closure

The following material is provided as clarification for the subject of shutting down socket connections closing the sockets. 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, hereafter referred to as a shutdown sequence. Two general classes of shutdown sequences are defined: graceful and abortive (also called "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 by way of the socket option mechanism that allow the programmer to indicate whether the implicit shutdown sequence should be graceful or abortive, and also whether the **closesocket** function should linger (that is not complete immediately) to allow time for a graceful shutdown sequence to complete. These important distinctions and the ramifications of using **closesocket** in this manner have not been widely understood.

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, delaying 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, immediate return – allowing the shutdown sequence to complete in the background. Although this is the default behavior, the application has no way of knowing when (or whether) the graceful shutdown sequence actually completes.

One technique that can be used to minimize the chance of problems occurring during connection teardown is to avoid relying on an implicit shutdown being initiated by **closesocket**. Instead, use one of the two explicit shutdown functions, **shutdown** or **WSASendDisconnect**). 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 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.

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 can be delivered to the user out of the normal sequence. Examples include "expedited data" in X.25 and other OSI protocols, and "urgent data" in BSD Unix's use of TCP. The next section describes OOB data handling in a protocol-independent manner. A discussion of OOB data implemented using TCP "urgent data" follows it. In the each discussion, the use of **recv** also implies **recvfrom**, **WSARecv**, and **WSARecvFrom**, and references to **WSAAsyncSelect** also apply to **WSAEventSelect**.

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 can contain at least one byte of data, and at least one OOB data block can be pending delivery to the user at any one time. For communications protocols that support in-band signaling (such as 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 such as 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 might process out-of-band data "in-line", as part of the normal data stream. This is achieved by setting the socket option SO_OOBINLINE with **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 to determine whether there is any unread OOB data preceding the mark. For example, it can 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):

- Windows Sockets 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 with no OOB data previously queued. The FD_OOB is also posted when data is read using the MSG_OOB flag while some OOB data remains queued after the read operation has returned. FD_READ messages are not posted for OOB data.
- Windows Sockets 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**, Windows Sockets notifies the application with FD_OOB or with *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. OOB data is treated as normal for the purpose of the **select** and **WSAAsyncSelect** functions, and indicated by setting the socket in *readfds* or by sending an FD_READ message respectively.
- The application can 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.

OOB data in TCP

> **Important** 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 that there are, at present, two conflicting interpretations of RFC 793 (where 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, 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 to interoperate with an existing service. Windows Sockets suppliers are urged to document the out-of-band semantics (BSD or RFC 1122) that their product implements.

Arrival of a TCP segment with the "URG" (for urgent) 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). It might, therefore, point to data that has not yet been received.

If SO_OOBINLINE is 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 can 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.

Summary of Windows Sockets 2 Functions

The following 2 tables summarize the functions included in Windows Sockets 2. The functions are sorted into Berkeley-style functions and Microsoft Windows-specific Extension functions.

Socket Functions

The Windows Sockets specification includes the following Berkeley-style socket routines:

accept[1]	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[1]	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[1]	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[2]	Convert a 32-bit quantity from host byte order to network byte order.
htons[2]	Convert a 16-bit quantity from host byte order to network byte order.
inet_addr[2]	Converts a character string representing a number in the Internet standard "." notation to an Internet address value.
inet_ntoa[2]	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[2]	Convert a 32-bit quantity from network byte order to host byte order.
ntohs[2]	Convert a 16-bit quantity from network byte order to host byte order.
recv[1]	Receive data from a connected or unconnected socket.
recvfrom[1]	Receive data from either a connected or unconnected socket.
select[1]	Perform synchronous I/O multiplexing.
send[1]	Send data to a connected socket.
sendto[1]	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.

1 The routine can block if acting on a blocking socket.

2 The routine is retained for backward compatibility with Windows Sockets 1.1, and should only be used for sockets created with AF_INET address family.

Microsoft Windows-specific Extension Functions

The Windows Sockets 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 Windows Sockets 2, please see New Concepts, Additions and Changes for Windows Sockets 2.

WSAAccept[1]	An extended version of accept which allows for conditional acceptance and socket grouping.
WSAAsyncGetHostByAddr[2] **WSAAsyncGetHostByName**[2] **WSAAsyncGetProtoByName**[2] **WSAAsyncGetProtoByNumber**[2] **WSAAsyncGetServByName**[2] **WSAAsyncGetServByPort**[2]	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.
WSAAsyncSelect	Perform asynchronous version of select
WSACancelAsyncRequest[2]	Cancel an outstanding instance of a WSAAsyncGetXByY function.
WSACleanup	Sign off from the underlying Windows Sockets DLL.
WSACloseEvent	Destroys an event object.
WSAConnect[1]	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 Windows Sockets 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[1]	Overlapped-capable version of ioctl
WSAJoinLeaf[1]	Add a multipoint leaf to a multipoint session

WSANtohl	Extended version of ntohl
WSANtohs	Extended version of ntohs
WSARecv[1]	An extended version of recv which accommodates scatter/gather I/O, overlapped sockets and provides the *flags* parameter as IN OUT
WSARecvFrom[1]	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[1]	An extended version of send which accommodates scatter/gather I/O and overlapped sockets
WSASendTo[1]	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 Windows Sockets DLL.
WSAWaitForMultipleEvents[1]	Blocks on multiple event objects.

1 The routine can block if acting on a blocking socket.

2 The routine is always realized by the name resolution provider associated with the default TCP/IP service provider, if any.

Registration and Name Resolution

Windows Sockets 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, Windows Sockets 2 applications need not be cognizant of the widely differing interfaces associated with name services such as DNS, NIS, X.500, SAP, etc. To maintain full backward compatibility with Windows Sockets 1.1, all of the existing **getXbyY** and asynchronous **WSAAsyncGetXbyY** database lookup functions continue to be supported, but are implemented in the Windows Sockets service provider interface in terms of the new name resolution capabilities. For more information, see Windows Sockets 1.1 Compatibile Name Resolution for TCP/IP.

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 Windows Sockets 2 interface adopts the model described below.

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 Windows Sockets name resolution.

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.

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.

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 Windows Sockets'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:

{ewc msdncd, EWGraphic, bsd23511 0 /a "SDK_1.WMF"}

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 Windows Sockets, 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 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 Windows Sockets, 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 WS2_32.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 WS2_32.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 WS2_32.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 Windows Sockets 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. 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.

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.

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.

Client Query

- **WSAEnumNameSpaceProviders**
- **WSALookupServiceBegin**
- **WSALookupServiceNext**
- **WSALookupServiceEnd**

The **WSAEnumNameSpaceProviders** function allows an application to discover which name spaces are accessible via Windows Sockets'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.

Windows Sockets' 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.

Helper Functions

- **WSAGetServiceClassNameByServiceClassId**

- **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
IS_SVCID_TCP(GUID) IS_SVCID_UDP(GUID) IS_SVCID_NETWARE(GUID)	Returns TRUE if the GUID is within the allowable range
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

Name Resolution Data Structures

There are several important data structures that are used extensively throughout the name resolution functions. These are described below.

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:

{ewc msdncd, EWGraphic, bsd23511 1 /a "SDK_2.WMF"}

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.

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.

{ewc msdncd, EWGraphic, bsd23511 2 /a "SDK_3.WMF"}

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.

Windows Sockets 1.1 Compatibile Name Resolution for TCP/IP

Windows Sockets 1.1 defined a number of routines that were used for name resolution with TCP/IP networks. These are customarily called 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
WSAAsyncGetServByPort

There are also two functions (now implemented in the WinSock 2 DLL) used to convert dotted internet address notation to and from string and binary representations, respectively:

inet_addr
inet_ntoa

All of these functions are specific to 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 Windows Sockets 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.

The WS2_32.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. The WS2_32.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.

Basic Approach for getxbyy

Most getXbyY functions are translated by the WS2_32.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 WS2_32.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 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.

The getprotobyname and getprotobynumber functions

These functions are implemented within the WS2_32.DLL by consulting a local protocols database. They do not result in any name resolution query.

The getservbyname and getservbyport functions

The **WSAServiceLookupBegin** 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 WS2_32.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 structure 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 GUIDs from svcs.h that corresponds to the port number indicated in the servent structure.

The gethostbyname function

The **WSAServiceLookupBegin** query uses SVCID_INET_HOSTADDRBYNAME as the service class GUID. The host name is supplied in *lpszServiceInstanceName*. The WS2_32.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.

The gethostbyaddr function

The **WSAServiceLookupBegin** 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 WS2_32.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.

The gethostname function

The **WSAServiceLookupBegin** query uses SVCID_HOSTNAME as the service class GUID. If *lpszServiceInstanceName* is NULL or references a NULL string (that is . ""), the local host is to be resolved. Otherwise, a lookup on a specified host name shall occur. For the purposes of emulating **gethostname** the WS2_32.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.

Multipoint and Multicast Semantics

In considering how to support multipoint and multicast in Windows Sockets 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.

Multipoint Taxonomy

The taxonomy described in this section 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 Windows Sockets 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 undsi-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 Windows Sockets 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)

Windows Sockets 2 Interface Elements for Multipoint and Multicast

The mechanisms that have been incorporated into Windows Sockets 2 for utilizing multipoint capabilities can be summarized as follows:

Three attribute bits in the WSAPROTOCOL_INFO struct

1. Four flags defined for the *dwFlags* parameter of **WSASocket**
2. One function, **WSAJoinLeaf**, for adding leaf nodes into a multipoint session
3. 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.

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.

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".

SIO_MULTIPOINT_LOOPBACK 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_LOOPBACK command code for **WSAIoctl** is used to enable or disable loopback of multipoint traffic.

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.

The function **WSAJoinLeaf** is used to join a leaf node into the multipoint session. See below for a discussion on how this function is utilized.

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.

(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.)

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)

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 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.

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.

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 Windows Sockets 2 multipoint functions. We chose these two as examples because they are very popular and well understood.

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 Windows Sockets 2 Protocol Specific Annex.

The remaining three operations are covered well with the Windows Sockets 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_LOOPBACK 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.

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.

Additional Windows Socket Information

This section contains information on the Windows Sockets 2 header file, additional Windows Sockets reference material, and the error codes encountered in programming for Windows Sockets 2.

Windows Sockets 2 Header File - WINSOCK2.H

New versions of WINSOCK2.H will appear periodically as new identifiers are allocated by the Windows Sockets 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.

Additional Documentation

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:

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.*

Petzold, C. [1992], *Programming Windows 3.1,* Microsoft Press, Redmond, Washington.

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

Windows Sockets Books

Bonner, P. [1995], *Network Programming with Windows Sockets*, ISBN: 0-13-230152-0, Prentice Hall, Englewood Cliffs, New Jersey.

Dumas, A. [1995], *Programming Windows Sockets,* 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.

Error Codes

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.

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, or **connect**/**sendto** an address or port that is not valid for a remote machine (e.g. 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** for more information on the **SO_LINGER** option on the remote socket.)

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 host you were connected to crashed and rebooted. 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.

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 Windows Sockets 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 2.0.

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_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 **WSAWaitForMultipleEvents** in a polling mode to determine when an overlapped operation has completed will get this error code until the operation is complete.

WSA_NOT_ENOUGH_MEMORY

(OS dependent)

Insufficient memory available.

An application used a Windows Sockets 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 functions fails trying to manipulate theprotocol/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 Windows Sockets 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 Windows Sockets 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 Windows Sockets 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
(10094)

Graceful shutdown in progress.
Returned by **recv**, **WSARecv** 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**.

accept Quick Info

The Windows Sockets **accept** function accepts a connection on a socket.

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

Parameters

s

[in] A descriptor identifying a socket which is listening for connections after a **listen**.

addr

[out] 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

[out] 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 nonblocking, **accept** blocks the caller until a connection is present. If the socket is marked nonblocking and no pending connections are present on the queue, **accept** returns an error as described below. The accepted socket cannot be used to accept more connections. The original socket remains open.

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 Values

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 can 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 FUNCTION.
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	The (blocking) call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking Windows Sockets 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 nonempty 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 nonblocking and no connections are present to be accepted.

See Also

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

AcceptEx Quick Info

Notice This function is a Microsoft-specific extension to the Windows Sockets specification. For more information, see Microsoft Extensions and Windows Sockets 2.

The Windows Sockets **AcceptEx** function accepts a new connection, returns the local and remote address, and receives the first block of data sent by the client application.

```
BOOL AcceptEx (
  SOCKET sListenSocket,
  SOCKET sAcceptSocket,
  PVOID lpOutputBuffer,
  DWORD dwReceiveDataLength,
  DWORD dwLocalAddressLength,
  DWORD dwRemoteAddressLength,
  LPDWORD lpdwBytesReceived,
  LPOVERLAPPED lpOverlapped
);
```

Parameters

sListenSocket
[in] A descriptor identifying a socket that has already been called with the **listen** function. A server application waits for attempts to connect on this socket.

sAcceptSocket
[in] A descriptor identifying a socket on which to accept an incoming connection. This socket must not be bound or connected.

lpOutputBuffer
[in] A pointer to a buffer that receives the first block of data sent on a new connection, the local address of the server, and the remote address of the client. The receive data is written to the first part of the buffer starting at offset zero, while the addresses are written to the latter part of the buffer. This parameter must be specified.

dwReceiveDataLength
[in] The number of bytes in the buffer that will be used for receiving data. If this parameter is specified as zero, then no receive operation is performed in conjunction with accepting the connection. Instead, the **AcceptEx** function completes as soon as a connection arrives without waiting for any data.

dwLocalAddressLength
[in] The number of bytes reserved for the local address information. This must be at least 16 bytes more than the maximum address length for the transport protocol in use.

dwRemoteAddressLength
[in] The number of bytes reserved for the remote address information. This must be at least 16 bytes more than the maximum address length for the transport protocol in use.

lpdwBytesReceived
[out] A pointer to a DWORD that receives the count of bytes received. This is set only if the operation completes synchronously. If it returns ERROR_IO_PENDING and is completed later, then this DWORD is never set and you must obtain the number of bytes read from the completion notification mechanism.

lpOverlapped
[in] An OVERLAPPED structure that is used to process the request. This parameter *must* be specified; it cannot be NULL.

Return Values

If no error occurs, the **AcceptEx** function was successfully completed and a value of TRUE is returned. If the function was unsuccesful, the **AcceptEx** function returns FALSE. The **GetLastError** function can then

be called to return extended error information. If **GetLastError** returns ERROR_IO_PENDING, then the operation was successfully initiated and is still in progress.

Remarks

The **AcceptEx** function combines several socket functions into a single API/kernel transition. The **AcceptEx** function, when successful, performs three tasks: a new connection is accepted, both the local and remote addresses for the connection are returned, and the first block of data sent by the remote is received. A program will make a connection to a socket more quickly using the **AcceptEx** function instead of the **Accept** function.

A single output buffer receives the data, the local socket address (the server), and the remote socket address (the client). Using a single buffer improves performance, but the **GetAcceptExSockaddrs** function must be called to parse the buffer into its three distinct parts.

Because the addresses are written in an internal format, the buffer size for the local and remote address must be 16 bytes more than the size of the SOCKADDR structure for the transport protocol in use. For example, the size of a SOCKADDR_IN − the address structure for TCP/IP − is 16 bytes, so specify a buffer size of at least 32 bytes for the local and remote addresses.

The **AcceptEx** function uses overlapped I/O, unlike the Windows Sockets 1.1 **accept** function. If your application uses the **AcceptEx** function, it can service a large number of clients with a relatively small number of threads.

As with all overlapped Win32 functions, either Win32 events or completion ports can be used as a completion notification mechanism.

Another key difference between the **AcceptEx** function and the Windows Sockets 1.1 **accept** function is that the **AcceptEx** function requires the caller to already have two sockets: one that specifies the socket on which to listen and one that specifies the socket on which to accept the connection. The *sAcceptSocket* parameter must be an open socket that is neither bound nor connected.

The *lpNumberOfBytesTransferred* parameter of the **GetQueuedCompletionStatus** function or the **GetOverlappedResult** function indicates the number of bytes received in the request.

When this operation is successfully completed, *sAcceptHandle* can be passed only to the following functions:

ReadFile
WriteFile
send
recv
TransmitFile
closesocket

> **Note** If you have called the **TransmitFile** function with both the TF_DISCONNECT and TF_REUSE_SOCKET flags, the specified socket has been returned to a state in which it is neither bound nor connected. You can then pass the handle of the socket to the **AcceptEx** function in the *sAcceptSocket* parameter.

In order to use *sAcceptHandle* with other Window Sockets 1.1 functions, call the **setsockopt** function with the SO_UPDATE_ACCEPT_CONTEXT option. This option initializes the socket so that other Windows Sockets routines to access the socket correctly.

When the **AcceptEx** function returns, *sAcceptSocket* is in the default state for a connected socket. The socket associated with the *sAcceptSocket* parameter does not inherit the properties of the socket associated with *sListenSocket* parameter until SO_UPDATE_ACCEPT_CONTEXT is set on the socket.

Use the **setsockopt** function to set the SO_UPDATE_ACCEPT_CONTEXT option, specifying *sAcceptSocket* as the socket handle and *sListenSocket* as the option value.

For example:

```
err = setsockopt( sAcceptSocket,
    SOL_SOCKET,
    SO_UPDATE_ACCEPT_CONTEXT,
    (char *)&sListenSocket,
    sizeof(sListenSocket) );
```

You can use the **getsockopt** function with the SO_CONNECT_TIME option to check whether a connection has been accepted. If it has been accepted, you can determine how long the connection has been established. The return value is the number of seconds that the socket has been connected. If the socket is not connected, the **getsockopt** returns 0xFFFFFFFF. Checking a connection like this is necessary in order to check for connections that have been established for awhile, but no data has been received. You can then kill those connections.

For example:

```
INT seconds;
INT bytes = sizeof(seconds);

err = getsockopt( sAcceptSocket, SOL_SOCKET, SO_CONNECT_TIME,
                      (char *)&seconds, (PINT)&bytes );
if ( err != NO_ERROR ) {
    printf( "getsockopt(SO_CONNECT_TIME) failed: %ld\n",
WSAGetLastError( ) );
    exit(1);
}
```

bind Quick Info

The Windows Sockets **bind** function associates a local address with a socket.

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

Parameters

s

[in] A descriptor identifying an unbound socket.

name

[in] 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.

namelen

[in] 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 Windows Sockets 2, the *name* parameter is not strictly interpreted as a pointer to a "sockaddr" structure. 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 can 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 multihomed hosts (that is, hosts that have more than one network interface and address). For TCP/IP, if the port is specified as zero, the service provider will assign a unique port to the application with a value between 1024 and 5000. The application can 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 can be valid if the host is multihomed.

Return Values

If no error occurs, **bind** returns zero. Otherwise, it returns SOCKET_ERROR, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur

	before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The specified address is already in use. (See the SO_REUSEADDR socket option under **setsockopt**.)
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 Windows Sockets 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, **getsockname**, **listen**, **setsockopt**, **socket**, **WSACancelBlockingCall**

closesocket Quick Info

The Windows Sockets **closesocket** function closes a socket.

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

Parameters

s

[in] 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, 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. 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	Nonzero	Graceful	Yes

If SO_LINGER is set (that is, the *l_onoff* field of the linger structure is nonzero; see Multipoint and Multicast Semantics) with a zero time-out 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 nonzero time-out interval on a blocking socket, the **closesocket** call blocks on a blocking socket until the remaining data has been sent or until the time-out expires. This is called a graceful disconnect. If the time-out expires before all data has been sent, the Windows Sockets implementation terminates the connection before **closesocket** returns.

Enabling SO_LINGER with a nonzero time-out interval on a nonblocking 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** can continue to fail unless the application disables SO_DONTLINGER, enables SO_LINGER with a zero time-out, or calls **shutdown** to initiate closure.

If SO_DONTLINGER is set on a stream socket (that is, the *l_onoff* field of the linger structure is zero; see Multipoint and Multicast Semantics) the **closesocket** call will return immediately. 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 Windows Sockets provider cannot release the socket and other resources for an arbitrary period, which can affect applications which expect to use all available

sockets. This is the default behavior.

Note To assure that all data is sent and received on a connection, an application should call **shutdown** before calling **closesocket** (see 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 - connection is gracefully closed "in the background"
- if SO_LINGER enabled with a zero time-out: it always returns immediately - connection is reset/terminated
- if SO_LINGER enabled with nonzero time-out:
  - with blocking socket it blocks until all data sent or time-out expires
  - with nonblocking socket it returns immediately indicating failure

For additional information please see Graceful shutdown, linger options and socket closure for more information.

Return Values

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

Error Codes

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

See Also

accept, **ioctlsocket**, **setsockopt**, **socket**, **WSAAsyncSelect**, **WSADuplicateSocket**

connect Quick Info

The Windows Sockets **connect** function establishes a connection to a peer.

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

Parameters

s
[in] A descriptor identifying an unconnected socket.

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

namelen
[in] 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 (for example, 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 a connectionless socket (for example, 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." Then, the default remote address will be indeterminate, so **send/WSASend** and **recv/WSARecv** calls will return the error code WSAENOTCONN. However, **sendto/WSASendTo** and **recvfrom/WSARecvFrom** can still be used. The default destination can 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* can 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 (that is, 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 Values

If no error occurs, **connect** returns zero. Otherwise, it returns SOCKET_ERROR, and a specific error code can be retrieved by calling **WSAGetLastError**.

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

With a nonblocking socket, the connection attempt cannot be completed immediately. In this case, **connect** will return SOCKET_ERROR, and **WSAGetLastError** will return WSAEWOULDBLOCK. In this case, the application can:

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, 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.

For a nonblocking socket, until the connection attempt completes all subsequent calls to **connect** on the same socket will fail with the error code WSAEALREADY.

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The specified address is already in use.
WSAEINTR	The (blocking) call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	A nonblocking **connect** call is in progress on the specified socket. **Note** In order to preserve backward compatibility, this error is reported as WSAEINVAL to Windows Sockets 1.1 applications that link to either WINSOCK.DLL or WSOCK32.DLL.
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
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 cannot 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 nonblocking and the connection cannot be completed immediately. It is possible to **select** the socket while it is connecting by **select**ing it for writing.
WSAEACCES	Attempt to connect datagram socket to broadcast address failed because **setsockopt** SO_BROADCAST is not enabled.

See Also

accept, **bind**, **getsockname**, **select**, **socket**, **WSAAsyncSelect**, **WSAConnect**

EnumProtocols Quick Info

Important The **EnumProtocols** function is a Microsoft-specific extension to the Windows Sockets 1.1 specification. This function is obsolete. For the convenience of Windows Sockets 1.1 developers, the reference material is below.

In Windows Sockets 2, this functionality is realized with the function **WSAEnumProtocols**.

The **EnumProtocols** function obtains information about a specified set of

network protocols that are active on a local host.

```
INT EnumProtocols(
  LPINT lpiProtocols,              // pointer to array of protocol identifiers
  LPVOID lpProtocolBuffer,         // pointer to buffer to receive protocol information
  LPDWORD lpdwBufferLength         // pointer to variable that specifies the size of the receiving buffer
);
```

Parameters

lpiProtocols

Pointer to a null-terminated array of protocol identifiers. The **EnumProtocols** function obtains information about the protocols specified by this array.

If *lpiProtocols* is NULL, the function obtains information about all available protocols.

The following protocol identifier values are defined:

Value	Protocol
IPPROTO_TCP	TCP/IP, a connection/stream-oriented protocol
IPPROTO_UDP	User Datagram Protocol (UDP/IP), a connectionless datagram protocol
ISOPROTO_TP4	ISO connection-oriented transport protocol
NSPROTO_IPX	IPX
NSPROTO_SPX	SPX
NSPROTO_SPXII	SPX II

lpProtocolBuffer

Pointer to a buffer that the function fills with an array of **PROTOCOL_INFO** data structures.

lpdwBufferLength

Pointer to a variable that, on input, specifies the size, in bytes, of the buffer pointed to by *lpProtocolBuffer*.

On output, the function sets this variable to the minimum buffer size needed to retrieve all of the requested information. For the function to succeed, the buffer must be at least this size.

Return Values

If the function succeeds, the return value is the number of **PROTOCOL_INFO** data structures written to the buffer pointed to by *lpProtocolBuffer*.

If the function fails, the return value is **SOCKET_ERROR** (- 1). To get extended error information, call **GetLastError**. **GetLastError** may return the following extended error code:

Value	Meaning
ERROR_INSUFFICIENT_BUFFER	The buffer pointed to by *lpProtocolBuffer* was too small to

receive all of the relevant **PROTOCOL_INFO** structures. Call the function with a buffer at least as large as the value returned in **lpdwBufferLength*.

Remarks

In the following sample code, the **EnumProtocols** function obtains information about all protocols that are available on a system. The code then examines each of the protocols in greater detail.

```
SOCKET
OpenConnection (
    PTSTR ServiceName,
    PGUID ServiceType,
    BOOL Reliable,
    BOOL MessageOriented,
    BOOL StreamOriented,
    BOOL Connectionless,
    PINT ProtocolUsed
    )
{
    // local variables
    INT protocols[MAX_PROTOCOLS+1];
    BYTE buffer[2048];
    DWORD bytesRequired;
    INT err;
    PPROTOCOL_INFO protocolInfo;
    PCSADDR_INFO csaddrInfo;
    INT protocolCount;
    INT addressCount;
    INT i;
    DWORD protocolIndex;
    SOCKET s;

    // First look up the protocols installed on this machine.
    //
    bytesRequired = sizeof(buffer);
    err = EnumProtocols( NULL, buffer, &bytesRequired );
    if ( err <= 0 )
        return INVALID_SOCKET;

    // Walk through the available protocols and pick out the ones which
    // support the desired characteristics.
    //
    protocolCount = err;
    protocolInfo = (PPROTOCOL_INFO)buffer;

    for ( i = 0, protocolIndex = 0;
        i < protocolCount && protocolIndex < MAX_PROTOCOLS;
        i++, protocolInfo++ ) {

        // If connection-oriented support is requested, then check if
        // supported by this protocol.  We assume here that connection-
        // oriented support implies fully reliable service.
        //
```

```
if ( Reliable ) {
    // Check to see if the protocol is reliable.  It must
    // guarantee both delivery of all data and the order in
    // which the data arrives.
    //
    if ( (protocolInfo->dwServiceFlags &
            XP_GUARANTEED_DELIVERY) == 0
        ||
            (protocolInfo->dwServiceFlags &
            XP_GUARANTEED_ORDER) == 0 ) {

        continue;
        }

    // Check to see that the protocol matches the stream/message
    // characteristics requested.
    //
    if ( StreamOriented &&
        (protocolInfo->dwServiceFlags & XP_MESSAGE_ORIENTED)
            != 0 &&
        (protocolInfo->dwServiceFlags & XP_PSEUDO_STREAM)
             == 0 ) {
        continue;
        }

    if ( MessageOriented &&
            (protocolInfo->dwServiceFlags & XP_MESSAGE_ORIENTED)
                      == 0 ) {
        continue;
        }

    }
else if ( Connectionless ) {
    // Make sure that this is a connectionless protocol.
    //
    if ( (protocolInfo->dwServiceFlags & XP_CONNECTIONLESS)
             != 0 )
        continue;
    }

// This protocol fits all the criteria.  Add it to the list of
// protocols in which we're interested.
//
protocols[protocolIndex++] = protocolInfo->iProtocol;
}
```

See Also

GetAddressByName, PROTOCOL_INFO

GetAcceptExSockaddrs Quick Info

Notice This function is a Microsoft-specific extension to the Windows Sockets specification. For more information, see Microsoft Extensions and Windows Sockets 2.

The Windows Sockets **GetAcceptExSockaddrs** function parses the data obtained from a call to the **AcceptEx** function and passes the local and remote addresses to a SOCKADDR structure.

```
VOID GetAcceptExSockaddrs (
  PVOID lpOutputBuffer,
  DWORD dwReceiveDataLength,
  DWORD dwLocalAddressLength,
  DWORD dwRemoteAddressLength,
  LPSOCKADDR *LocalSockaddr,
  LPINT LocalSockaddrLength,
  LPSOCKADDR *RemoteSockaddr,
  LPINT RemoteSockaddrLength
);
```

Parameters

lpOutputBuffer
[in] A pointer to a buffer that receives the first block of data sent on a connection resulting from an **AcceptEx** call. It must be the same *lpOutputBuffer* parameter that was passed to the **AcceptEx** function.

dwReceiveDataLength
[in] The number of bytes in the buffer that will be used for receiving the first data. This must be equal to the *dwReceiveDataLength* parameter that was passed to the **AcceptEx** function.

dwLocalAddressLength
[in] The number of bytes reserved for the local address information. This must be equal to the *dwLocalAddressLength* parameter that was passed to the **AcceptEx** function.

dwRemoteAddressLength
[in] The number of bytes reserved for the remote address information. This must be equal to the *dwRemoteAddressLength* parameter that was passed to the **AcceptEx** function.

LocalSockaddr
[out] A pointer to the SOCKADDR structure that receives the local address of the connection (the same information that would be returned by the Windows Sockets **getsockname** function). This parameter must be specified.

LocalSockaddrLength
[out] The size of the local address. This parameter must be specified.

RemoteSockaddr
[out] A pointer to the SOCKADDR structure that receives the remote address of the connection (the same information that would be returned by the Windows Sockets **getpeername** function). This parameter must be specified.

RemoteSockaddrLength
[out] The size of the local address. This parameter must be specified.

Return Values

This function does not return a value.

Remarks

The **GetAcceptExSockaddrs** function is used exclusively with the **AcceptEx** function to parse the first data that the socket receives into local and remote addresses. You are required to use this function

because the **AcceptEx** function writes address information in an internal (TDI) format. The **GetAcceptExSockaddrs** routine is required to locate the SOCKADDR structures in the buffer.

GetAddressByName Quick Info

Important The **GetAddressByName** function is a Microsoft-specific extension to the Windows Sockets 1.1 specification. This function is obsolete. For the convenience of Windows Sockets 1.1 developers, the reference material is below.

In Windows Sockets 2, this functionality is realized with the functions detailed in *Protocol-Independent Name Resolution*.

The **GetAddressByName** function queries a name space, or a set of default name spaces, in order to obtain network address information for a specified network service. This process is known as service name resolution. A network service can also use the function to obtain local address information that it can use with the **bind** function.

```
INT GetAddressByName(
  DWORD dwNameSpace,                          // name space to query for service address information
  LPGUID lpServiceType,                       // the type of the service
  LPTSTR lpServiceName,                       // the name of the service
  LPINT lpiProtocols,                         // points to array of protocol identifiers
  DWORD dwResolution,                         // set of bit flags that specify aspects of name resolution
  LPSERVICE_ASYNC_INFO lpServiceAsyncInfo,    // reserved for future use, must be NULL
  LPVOID lpCsaddrBuffer,                      // points to buffer to receive address information
  LPDWORD lpdwBufferLength,                   // points to variable with address buffer size information
  LPTSTR lpAliasBuffer,                       // points to buffer to receive alias information
  LPDWORD lpdwAliasBufferLength               // points to variable with alias buffer size information
);
```

Parameters

dwNameSpace

Specifies the name space, or a set of default name spaces, that the operating system will query for network address information.

Use one of the following constants to specify a name space:

Value	Name Space
NS_DEFAULT	A set of default name spaces. The function queries each name space within this set. The set of default name spaces typically includes all the name spaces installed on the system. System administrators, however, can exclude particular name spaces from the set. This is the value that most applications should use for *dwNameSpace*.
NS_DNS	The Domain Name System used in the Internet for host name resolution.
NS_NETBT	The NetBIOS over TCP/IP layer. All Windows NT systems register their computer names with NetBIOS. This name space is used to convert a computer name to an IP address that uses this registration. Note that NS_NETBT may access a WINS server to perform the resolution.
NS_SAP	The Netware Service Advertising

	Protocol. This may access the Netware bindery if appropriate. NS_SAP is a dynamic name space that allows registration of services.
NS_TCPIP_HOSTS	Lookup value in the <systemroot>\system32\drivers\etc\hosts file.
NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and looks up host names and IP addresses in cache of host to IP address mappings.

Most calls to **GetAddressByName** should use the special value NS_DEFAULT. This lets a client get by with no knowledge of which name spaces are available on an internetwork. The system administrator determines name space access. Name spaces can come and go without the client having to be aware of the changes.

lpServiceType

Points to a globally unique identifier (**GUID**) that specifies the type of the network service. The header file SVCGUID.H includes definitions of several **GUID** service types, and macros for working with them.

lpServiceName

Points to a zero-terminated string that uniquely represents the service name. For example, "MY SNA SERVER".

Setting *lpServiceName* to NULL is the equivalent of setting *dwResolution* to RES_SERVICE. The function operates in its second mode, obtaining the local address to which a service of the specified type should bind. The function stores the local address within the **LocalAddr** member of the **CSADDR_INFO** structures stored into **lpCsaddrBuffer*.

If *dwResolution* is set to RES_SERVICE, the function ignores the *lpServiceName* parameter.

If *dwNameSpace* is set to NS_DNS, **lpServiceName* is the name of the host.

lpiProtocols

Points to a zero-terminated array of protocol identifiers. The function restricts a name resolution attempt to name space providers that offer these protocols. This lets the caller limit the scope of the search.

If *lpiProtocols* is NULL, the function obtains information on all available protocols.

dwResolution

A set of bit flags that specify aspects of the service name resolution process. The following bit flags are defined:

Value	**Meaning**
RES_SERVICE	If this flag is set, the function obtains the address to which a service of the specified type should bind. This is the equivalent of setting *lpServiceName* to NULL. If this flag is clear, normal name resolution occurs.
RES_FIND_MULTIPLE	If this flag is set, the operating system performs an extensive search of all name spaces for the service. It will ask every appropriate name space to resolve the service name. If this flag is clear, the operating system stops looking for service addresses as soon as one is

	found.
RES_SOFT_SEARCH	This flag is valid if the name space supports multiple levels of searching. If this flag is valid and set, the operating system performs a simple and quick search of the name space. This is useful if an application only needs to obtain easy-to-find addresses for the service. If this flag is valid and clear, the operating system performs a more extensive search of the name space.

lplpServiceAsyncInfo
Reserved for future use; must be set to NULL.

lpCsaddrBuffer
Points to a buffer to receive one or more **CSADDR_INFO** data structures. The number of structures written to the buffer depends on the amount of information found in the resolution attempt. You should assume that multiple structures will be written, although in many cases there will only be one.

lpdwBufferLength
Points to a variable that, upon input, specifies the size, in bytes, of the buffer pointed to by *lpCsaddrBuffer*.

Upon output, this variable contains the total number of bytes required to store the array of **CSADDR_INFO** structures. If this value is less than or equal to the input value of **lpdwBufferLength*, and the function is successful, this is the number of bytes actually stored in the buffer. If this value is greater than the input value of **lpdwBufferLength*, the buffer was too small, and the output value of **lpdwBufferLength* is the minimal required buffer size.

lpAliasBuffer
Points to a buffer to receive alias information for the network service.

If a name space supports aliases, the function stores an array of zero-terminated name strings into the buffer pointed to by *lpAliasBuffer*. There is a double zero-terminator at the end of the list. The first name in the array is the service's primary name. Names that follow are aliases. An example of a name space that supports aliases is DNS.

If a name space does not support aliases, it stores a double zero-terminator into the buffer.

This parameter is optional, and can be set to NULL.

lpdwAliasBufferLength
Points to a variable that, upon input, specifies the size, in bytes, of the buffer pointed to by *lpAliasBuffer*.

Upon output, this variable contains the total number of bytes required to store the array of name strings. If this value is less than or equal to the input value of **lpdwAliasBufferLength*, and the function is successful, this is the number of bytes actually stored in the buffer. If this value is greater than the input value of **lpdwAliasBufferLength*, the buffer was too small, and the output value of **lpdwAliasBufferLength* is the minimal required buffer size.

If *lpAliasBuffer* is NULL, *lpdwAliasBufferLength* is meaningless and can also be NULL.

Return Values

If the function succeeds, the return value is the number of **CSADDR_INFO** data structures written to the buffer pointed to by *lpCsaddrBuffer*.

If the function fails, the return value is SOCKET_ERROR(- 1). To get extended error information, call **GetLastError**. **GetLastError** may return the following extended error value:

Value	Meaning

ERROR_INSUFFICIENT_BUFFER	The buffer pointed to by *lpCsaddrBuffer* was too small to receive all of the relevant **CSADDR_INFO** structures. Call the function with a buffer at least as large as the value returned in **lpdwBufferLength*.

Remarks

This function is a more powerful version of the Windows Sockets function **gethostbyname** The **GetAddressByName** function works with multiple name services.

The **GetAddressByName** function lets a client obtain a Windows Sockets address for a network service. The client specifies the service of interest by its service type and service name.

Many name services support a default prefix or suffix that the name service provider considers when resolving service names. For example, in the DNS name space, if a domain is named "nt.microsoft.com", and "ftp millikan" is provided as input, the DNS software fails to resolve "millikan", but successfully resolves "millikan.nt.microsoft.com".

Note that the **GetAddressByName** function can search for a service address in two ways: within a particular name space, or within a set of default name spaces. Using a default name space, an administrator can specify that certain name spaces will be searched for service addresses only if specified by name. An administrator or name space setup program can also control the ordering of name space searches.

See Also

gethostbyname, **CSADDR_INFO**

gethostbyaddr Quick Info

The Windows Sockets **gethostbyaddr** function gets host information corresponding to an address.

```
struct hostent FAR * gethostbyaddr (
  const char FAR * addr,
  int len,
  int type
);
```

Parameters

addr
[in] A pointer to an address in network byte order.

len
[in] The length of the address.

type
[in] The type of the address.

Remarks

gethostbyaddr returns a pointer to the following hostent structure which contains the name(s) and address which correspond to the given address. All strings are null terminated.

Return Values

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 can be retrieved by calling **WSAGetLastError**.

Error Codes

WSANOTINITIALISED	A successful WSAStartup must occur before using this function.
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	Nonrecoverable error occurred.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 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	The (blocking) call was canceled through **WSACancelBlockingCall**.

See Also

gethostbyname, **hostent**, **WSAAsyncGetHostByAddr**

gethostbyname Quick Info

The Windows Sockets **gethostbyname** function gets host information corresponding to a hostname.

```
struct hostent FAR * gethostbyname (
  const char FAR * name
);
```

Parameters

name
[out] A pointer to the null terminated name of the host.

Remarks

gethostbyname returns a pointer to a hostent structure. The contents of this structure correspond to the hostname *name*.

The pointer which is returned points to a structure which is allocated by Windows Sockets. 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 Windows Sockets function 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 Values

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 can be retrieved by calling **WSAGetLastError**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
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	Nonrecoverable error occurred.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 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	The (blocking) call was canceled through **WSACancelBlockingCall**.

See Also

gethostbyaddr, **WSAAsyncGetHostByName**

gethostname Quick Info

The Windows Sockets **gethostname** function returns the standard host name for the local machine.

```
int gethostname (
  char FAR * name,
  int namelen
);
```

Parameters

name

[out] A pointer to a buffer that will receive the host name.

namelen

[in] 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 Windows Sockets provider – it can be a simple host name, or it can 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 Values

If no error occurs, **gethostname** returns zero. Otherwise, it returns SOCKET_ERROR and a specific error code can 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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.

See Also

gethostbyname, **WSAAsyncGetHostByName**

GetNameByType Quick Info

Important The **GetNameByType** function is a Microsoft-specific extension to the Windows Sockets 1.1 specification. This function is obsolete. For the convenience of Windows Sockets 1.1 developers, the reference material is below.

In Windows Sockets 2, this functionality is realized with the functions detailed in *Protocol-Independent Name Resolution*.

The **GetNameByType** function obtains the name of a network service. The network service is specified by its service type.

```
INT GetNameByType(
  LPGUID lpServiceType,      // points to network service type GUID
  LPTSTR lpServiceName,      // points to buffer to receive name of network service
  DWORD dwNameLength         // points to variable that specifies buffer size
);
```

Parameters

lpServiceType

Points to a globally unique identifier (**GUID**) that specifies the type of the network service. The header file SVCGUID.H includes definitions of several **GUID** service types, and macros for working with them.

lpServiceName

Points to a buffer to receive a zero-terminated string that uniquely represents the name of the network service.

dwNameLength

Points to a variable that, on input, specifies the size of the buffer pointed to by *lpServiceName*. On output, the variable contains the actual size of the service name string.

Return Values

If the function succeeds, the return value is not SOCKET_ERROR (-1).

If the function fails, the return value is SOCKET_ERROR (-1). To get extended error information, call **GetLastError**.

See Also

GetTypeByName

getpeername Quick Info

The Windows Sockets **getpeername** function gets the address of the peer to which a socket is connected.

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

Parameters

s
[in] A descriptor identifying a connected socket.

name
[out] The structure which is to receive the name of the peer.

namelen
[out] 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 can 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 return, the *namelen* argument contains the actual size of the name returned in bytes.

Return Values

If no error occurs, **getpeername** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
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 Windows Sockets 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, **getsockname**, **socket**

getprotobyname Quick Info

The Windows Sockets **getprotobyname** function gets protocol information corresponding to a protocol name.

```
struct protoent FAR * getprotobyname (
  const char FAR * name
);
```

Parameters

name

[in] 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 Windows Sockets 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 Windows Sockets function calls.

Return Values

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 can be retrieved by calling **WSAGetLastError**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
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	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

WSAEINPROGRESS	A blocking Windows Sockets 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	The (blocking) call was canceled through **WSACancelBlockingCall**.

See Also

getprotobynumber, **WSAAsyncGetProtoByName**

getprotobynumber Quick Info

The Windows Sockets **getprotobynumber** function gets protocol information corresponding to a protocol number.

```
struct protoent FAR * getprotobynumber (
  int number
);
```

Parameters

number
[in] 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 Windows Sockets. 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 Windows Sockets function calls.

Return Values

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 can be retrieved by calling **WSAGetLastError**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
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	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	The (blocking) call was canceled through **WSACancelBlockingCall**.

See Also

getprotobyname, **WSAAsyncGetProtoByNumber**

getservbyname Quick Info

The Windows Sockets **getsservbyname** function gets service information corresponding to a service name and protocol.

```
struct servent FAR * getservbyname (
  const char FAR * name,
  const char FAR * proto
);
```

Parameters

name

[in] A pointer to a null terminated service name.

proto

[in] 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 can 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 Windows Sockets 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 Windows Sockets function calls.

Return Values

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 can be retrieved by calling **WSAGetLastError**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
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	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINTR	The (blocking) call was canceled through **WSACancelBlockingCall**.

See Also

getservbyport, **WSAAsyncGetServByName**

getservbyport Quick Info

The Windows Sockets **getservbyport** function gets service information corresponding to a port and protocol.

```
struct servent FAR * getservbyport (
  int port,
  const char FAR* proto
);
```

Parameters

port

[in] The port for a service, in network byte order.

proto

[in] 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 Windows Sockets. 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 Windows Sockets function calls.

Return Values

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 can be retrieved by calling **WSAGetLastError**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
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	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.
WSAEINPROGRESS	A blocking Windows Sockets 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	The (blocking) call was canceled through **WSACancelBlockingCall**.

See Also

getservbyname, **WSAAsyncGetServByPort**

GetService Quick Info

Important The **GetService** function is a Microsoft-specific extension to the Windows Sockets 1.1 specification. This function is obsolete. For the convenience of Windows Sockets 1.1 developers, the reference material is below..

In Windows Sockets 2, this functionality is realized with the functions detailed in *Protocol-Independent Name Resolution*.

The **GetService** function obtains information about a network service in the context of a set of default name spaces or a specified name space. The network service is specified by its type and name. The information about the service is obtained as a set of **NS_SERVICE_INFO** data structures.

```
INT GetService(
  DWORD dwNameSpace,                  // specifies name space or spaces to search
  PGUID lpGuid,                       // points to a GUID service type
  LPTSTR lpServiceName,               // points to a service name
  DWORD dwProperties,                 // specifies service information to be obtained
  LPVOID lpBuffer,                    // points to buffer to receive service information
  LPDWORD lpdwBufferSize,             // points to size of buffer, size of service information
  LPSERVICE_ASYNC_INFO lpServiceAsyncInfo   // reserved for future use, must be NULL
);
```

Parameters

dwNameSpace

Specifies the name space, or a set of default name spaces, that the operating system will query for information about the specified network service.

Use one of the following constants to specify a name space:

Value	Name Space
NS_DEFAULT	A set of default name spaces. The operating system will query each name space within this set. The set of default name spaces typically includes all the name spaces installed on the system. System administrators, however, can exclude particular name spaces from the set. NS_DEFAULT is the value that most applications should use for *dwNameSpace*.
NS_DNS	The Domain Name System used in the Internet for host name resolution.
NS_NETBT	The NetBIOS over TCP/IP layer. All Windows NT systems register their computer names with NetBIOS. This name space is used to resolve a computer name into an IP address using this registration. Note that NS_NETBT may access a WINS server to perform the resolution.
NS_SAP	The Netware Service Advertising Protocol. This may access the Netware bindery if appropriate. NS_SAP is a dynamic name space that allows

	registration of services.
NS_TCPIP_HOSTS	Looks up host names and IP addresses in the <systemroot>\system32\drivers\etc\hosts file.
NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and looks up host names and IP addresses in cache of host to IP address mappings.

Most calls to **GetService** should use the special value NS_DEFAULT. This lets a client get by with no knowledge of which name spaces are available on an internetwork. The system administrator determines name space access. Name spaces can come and go without the client having to be aware of the changes.

lpGuid

Points to a globally unique identifier (**GUID**) that specifies the type of the network service. The header file SVCGUID.H includes **GUID** service types from many well-known services within the DNS and SAP name spaces.

lpServiceName

Points to a zero-terminated string that uniquely represents the service name. For example, "MY SNA SERVER".

dwProperties

A set of bit flags that specify the service information that the function obtains. Each of these bit flag constants, other than PROP_ALL, corresponds to a particular member of the **SERVICE_INFO** data structure. If the flag is set, the function puts information into the corresponding member of the data structures stored in **lpBuffer*. The following bit flags are defined:

Value	**Name Space**
PROP_COMMENT	If this flag is set, the function stores data in the **lpComment** member of the data structures stored in **lpBuffer*.
PROP_LOCALE	If this flag is set, the function stores data in the **lpLocale** member of the data structures stored in **lpBuffer*.
PROP_DISPLAY_HINT	If this flag is set, the function stores data in the **dwDisplayHint** member of the data structures stored in **lpBuffer*.
PROP_VERSION	If this flag is set, the function stores data in the **dwVersion** member of the data structures stored in **lpBuffer*.
PROP_START_TIME	If this flag is set, the function stores data in the **dwTime** member of the data structures stored in **lpBuffer*.
PROP_MACHINE	If this flag is set, the function stores data in the **lpMachineName** member of the data structures stored in **lpBuffer*.
PROP_ADDRESSES	If this flag is set, the function stores data in the **lpServiceAddress** member of the data structures stored in **lpBuffer*.
PROP_SD	If this flag is set, the function stores data in the **ServiceSpecificInfo** member of the data structures stored in **lpBuffer*.
PROP_ALL	If this flag is set, the function stores data

in all of the members of the data structures stored in **lpBuffer*.

lpBuffer

Points to a buffer to receive an array of **NS_SERVICE_INFO** structures and associated service information. Each **NS_SERVICE_INFO** structure contains service information in the context of a particular name space. Note that if *dwNameSpace* is NS_DEFAULT, the function stores more than one structure into the buffer; otherwise, just one structure is stored.

Each **NS_SERVICE_INFO** structure contains a **SERVICE_INFO** structure. The members of these **SERVICE_INFO** structures will contain valid data based on the bit flags that are set in the *dwProperties* parameter. If a member's corresponding bit flag is not set in *dwProperties*, the member's value is undefined.

The function stores the **NS_SERVICE_INFO** structures in a consecutive array, starting at the beginning of the buffer. The pointers in the contained **SERVICE_INFO** structures point to information that is stored in the buffer between the end of the **NS_SERVICE_INFO** structures and the end of the buffer.

lpdwBufferSize

Points to a variable that, on input, contains the size, in bytes, of the buffer pointed to by *lpBuffer*. On output, this variable contains the number of bytes required to store the requested information. If this output value is greater than the input value, the function has failed due to insufficient buffer size.

lpServiceAsyncInfo

This parameter is reserved for future use. It must be set to NULL.

Return Values

If the function succeeds, the return value is the number of **NS_SERVICE_INFO** structures stored in **lpBuffer*. Zero indicates that no structures were stored.

If the function fails, the return value is SOCKET_ERROR (- 1). To get extended error information, call **GetLastError**. **GetLastError** may return one of the following extended error values:

Value	Meaning
ERROR_INSUFFICIENT_BUFFER	The buffer pointed to by *lpBuffer* is too small to receive all of the requested information. Call the function with a buffer at least as large as the value returned in **lpdwBufferSize*.
ERROR_SERVICE_NOT_FOUND	The specified service was not found, or the specified name space is not in use. The function return value is zero in this case.

See Also

SetService, **NS_SERVICE_INFO**, **SERVICE_INFO**

getsockname Quick Info

The Windows Sockets **getsockname** function gets the local name for a socket.

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

Parameters

s
[in] A descriptor identifying a bound socket.

name
[out] Receives the address (name) of the socket.

namelen
[out] 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 return, the *namelen* argument contains the actual size of the name returned in bytes.

If a socket was bound to an unspecified address (for example, 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 Windows Sockets application must not assume that the address will be specified unless the socket is connected. This is because for a multihomed host the address that will be used for the socket is unknown unless the socket is connected.

Return Values

If no error occurs, **getsockname** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
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 Windows Sockets 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, **getpeername**, **socket**

getsockopt Quick Info

The Windows Sockets **getsockopt** function retrieves a socket option.

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

Parameters

s
[in] A descriptor identifying a socket.

level
[in] 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
[in] The socket option for which the value is to be retrieved.

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

optlen
[in/out] 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 can exist at multiple protocol levels, but they are always present at the uppermost "socket" level. Options affect socket operations, such as the packet routing and out-of-band data transfer.

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*.

***level* = SOL_SOCKET**

Value	Type	Meaning
SO_ACCEPTCONN	BOOL	Socket is listening.
SO_BROADCAST	BOOL	Socket is configured for the transmission of broadcast messages.
SO_DEBUG	BOOL	Debugging is enabled.
SO_DONTLINGER	BOOL	If true, the SO_LINGER

		option is disabled.
SO_DONTROUTE	BOOL	Routing is disabled.
SO_ERROR	int	Retrieve error status and clear.
SO_GROUP_ID	GROUP	The identifier of the group to which this socket belongs.
SO_GROUP_PRIORITY	int	The relative priority for sockets that are part of a socket group.
SO_KEEPALIVE	BOOL	Keepalives are being sent.
SO_LINGER	struct linger	Returns the current linger options.
SO_MAX_MSG_SIZE	unsigned int	Maximum size of a message for message-oriented socket types (for example, SOCK_DGRAM). Has no meaning for stream-oriented sockets.
SO_OOBINLINE	BOOL	Out-of-band data is being received in the normal data stream. (See section Windows Sockets 1.1 Blocking Routines & EINPROGRESS for a discussion of this topic.)
SO_PROTOCOL_INFO	WSAPROTOCOL_INFO	Description of protocol info for protocol that is bound to this socket.
SO_RCVBUF	int	Buffer size for receives
SO_REUSEADDR	BOOL	The socket may be bound to an address which is already in use.
SO_SNDBUF	int	Buffer size for sends
SO_TYPE	int	The type of the socket (for example, SOCK_STREAM).
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.

***level* = IPPROTO_TCP**

TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.

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

Value	Type	Meaning
SO_RCVLOWAT	int	Receive low water mark
SO_RCVTIMEO	int	Receive time-out
SO_SNDLOWAT	int	Send low water mark
SO_SNDTIMEO	int	Send time-out
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
: Windows Sockets 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
: 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 give service provider. If this socket is not a group socket, the value is NULL.

SO_GROUP_PRIORITY
: 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 can 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 Windows Sockets 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 size of a message for message-

oriented socket types (for example, SOCK_DGRAM) as implemented by a particular service provider. It has no meaning for byte stream oriented sockets

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 can request different buffer sizes (larger or smaller). The call to **setsockopt** can 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 cannot be bound (see **bind**) to a local address which is already in use. On occasion, however, it may be necessary 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 Windows Sockets 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 can 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 Values

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
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 Windows Sockets 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**

GetTypeByName Quick Info

Important The **GetTypeByName** function is a Microsoft-specific extension to the Windows Sockets 1.1 specification. This function is obsolete. For the convenience of Windows Sockets 1.1 developers, the reference material is below.

In Windows Sockets 2, this functionality is realized with the functions detailed in *Protocol-Independent Name Resolution*.

The **GetTypeByName** function obtains a service type **GUID** for a network service specified by name.

```
INT GetTypeByName(
  LPTSTR lpServiceName,      // points to the name of the network service
  PGUID lpServiceType        // points to a variable to receive network service type
);
```

Parameters

lpServiceName

Points to a zero-terminated string that uniquely represents the name of the service. For example, "MY SNA SERVER".

lpServiceType

Points to a variable to receive a globally unique identifier (**GUID**) that specifies the type of the network service. The header file SVCGUID.H includes definitions of several **GUID** service types and macros for working with them.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is SOCKET_ERROR(- 1). To get extended error information, call **GetLastError**. **GetLastError** may return the following extended error value:

Value	Meaning
ERROR_SERVICE_DOES_NOT_EXIST	The specified service type is unknown.

See Also

GetNameByType

htonl Quick Info

The Windows Sockets **htonl** function converts a **u_long** from host to TCP/IP network byte order.

```
u_long htonl (
  u_long hostlong
);
```

Parameters

hostlong
[in] 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 Values

htonl returns the value in TCP/IP's network byte order.

See Also

htons, **ntohl**, **ntohs**, **WSAHtonl**, **WSAHtons**, **WSANtohl**, **WSANtohs**

htons Quick Info

The Windows Sockets **htons** function converts a **u_short** from host to TCP/IP network byte order.

```
u_short htons (
  u_short hostshort
);
```

Parameters

hostshort
[in] 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 Values

htons returns the value in TCP/IP network byte order.

See Also

htonl, **ntohl**, **ntohs**, **WSAHtonl**, **WSAHtons**, **WSANtohl**, **WSANtohs**

inet_addr Quick Info

The Windows Sockets **inet_addr** function converts a string containing an Internet Protocol dotted address into an **in_addr**.

```
unsigned long inet_addr (
  const char FAR * cp
);
```

Parameters

cp

[in] 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 Values

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

inet_ntoa Quick Info

The Windows Sockets **inet_ntoa** function converts a network address into a string in dotted format.

```
char FAR * inet_ntoa (
  struct in_addr in
);
```

Parameters

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 Windows Sockets. 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 Windows Sockets function call within the same thread, but no longer.

Return Values

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 Windows Sockets call is made.

See Also

inet_addr

ioctlsocket Quick Info

The Windows Sockets **ioctlsocket** function controls the mode of a socket.

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

Parameters

s

[in] A descriptor identifying a socket.

cmd

[in] The command to perform on the socket *s*.

argp

[in/out] A pointer to a parameter for *cmd*.

Remarks

This routine can 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. Here are the supported commands and their semantics:

FIONBIO

Enable or disable nonblocking mode on socket *s*. *argp* points at an **unsigned long**, which is nonzero if nonblocking mode is to be enabled and zero if it is to be disabled. When a socket is created, it operates in blocking mode (that is, nonblocking 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 zero, or disable **WSAEventSelect** by calling **WSAEventSelect** with the *lNetworkEvents* parameter equal to zero.

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 (for example, type SOCK_STREAM), FIONREAD returns an amount of data which can 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 (for example, 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 Windows Sockets 1.1 Blocking Routines & EINPROGRESS for a discussion of this topic.) This applies only to a socket of stream style (for example, 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 Values

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 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

getsockopt, **setsockopt**, **socket**, **WSAAsyncSelect**, **WSAEventSelect**, **WSAIoctl**

listen Quick Info

The Windows Sockets **listen** function establishes a socket to listen for an incoming connection.

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

Parameters

s

[in] A descriptor identifying a bound, unconnected socket.

backlog

[in] The maximum length to which the queue of pending connections can GROW. If this value is SOMAXCONN, then the underlying service provider responsible for socket *s* will set the backlog to a maximum "reasonable" value.

Remarks

To accept connections, a socket is first created with **socket**, 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, for example, 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 refill the queue to the current or most recent "backlog", if possible, and resume listening for incoming connections.

An application can 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

The *backlog* parameter is limited (silently) to a reasonable value as determined by the underlying service provider. Illegal values are replaced by the nearest legal value.

Return Values

If no error occurs, **listen** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	An attempt has been made to **listen** on an address in use.
WSAEINPROGRESS	A blocking Windows Sockets 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**

ntohl Quick Info

The Windows Sockets **ntohl** function converts a **u_long** from TCP/IP network order to host byte order.

```
u_long ntohl (
  u_long netlong
);
```

Parameters

netlong
[in] 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 Values

ntohl returns the value in host byte order.

See Also

htonl, **htons**, **ntohs**, **WSAHtonl**, **WSAHtons**, **WSANtohl**, **WSANtohs**

ntohs Quick Info

The Windows Sockets **ntohs** function converts a **u_short** from TCP/IP network byte order to host byte order.

```
u_short ntohs (
  u_short netshort
);
```

Parameters

netshort
[in] 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 Values

ntohs returns the value in host byte order.

See Also

htonl, **htons**, **ntohl**, **WSAHtonl**, **WSAHtons**, **WSANtohl**, **WSANtohs**

recv Quick Info

The Windows Sockets **recv** function receives data from a socket.

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

Parameters

s
[in] A descriptor identifying a connected socket.

buf
[out] A buffer for the incoming data.

len
[in] The length of *buf*.

flags
[in] 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.

For byte stream style socket (for example, 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 can use the **ioctlsocket** SIOCATMARK to determine whether any more out-of-band data remains to be read.

For message-oriented sockets (for example, type SOCK_DGRAM), data is extracted from the first enqueued datagram (message) from the destination address specified in the **connect** call. If unconnected, the socket must be bound, and there are no source address restrictions on datagrams received. 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 (for example, 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.

If no incoming data is available at the socket, the **recv** call waits for data to arrive unless the socket is nonblocking. In this case, a value of SOCKET_ERROR is returned with the error code set to WSAEWOULDBLOCK. The **select**, **WSAAsyncSelect**, or **WSAEventSelect** calls can 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 **recv** will complete immediately with zero bytes received. If the connection has been reset, a **recv** will fail with the error WSAECONNRESET.

The *flags* parameter can 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.
MSG_OOB	Process out-of-band data. (See section Out-Of-Band data for a discussion of this topic.)

Return Values

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
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	The (blocking) call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENETRESET	The connection has been broken due to the remote host resetting.
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 shut down; 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 nonblocking 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 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 zero or negative.
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
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 usable. On a UDP datagram socket this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.

See Also

recvfrom, **select**, **send**, **socket**, **WSAAsyncSelect**

recvfrom Quick Info

The Windows Sockets **recvfrom** function receives a datagram and stores the source address.

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

Parameters

s
: [in] A descriptor identifying a bound socket.

buf
: [out] A buffer for the incoming data.

len
: [in] The length of *buf*.

flags
: [in] Specifies the way in which the call is made.

from
: [out] An optional pointer to a buffer which will hold the source address upon return.

fromlen
: [in/out] An optional pointer to the size of the *from* buffer.

Remarks

This function is used to read incoming data on a (possibly connected) socket and capture the address from which the data was sent.

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 can use the **ioctlsocket** SIOCATMARK 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 (for example, UDP) the excess data is lost.

If *from* is nonzero, and the socket is not connection oriented (for example, 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 waits for data to arrive unless the socket is nonblocking. In this case, a value of SOCKET_ERROR is returned with the error code set to WSAEWOULDBLOCK. The **select**, **WSAAsyncSelect**, or **WSAEventSelect** can 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 zero bytes received. If the connection has been reset **recvfrom** will fail with the error WSAECONNRESET.

Flags can 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.
MSG_OOB	Process out-of-band data. (See section Out-Of-Band data for a discussion of this topic.)

Return Values

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
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	The (blocking) call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking Windows Sockets 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**, 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 zero or negative.
WSAENETRESET	The connection has been broken due to the remote host resetting.
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 send operations.
WSAESHUTDOWN	The socket has been shut down; 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 nonblocking and the **recvfrom** operation would block.
WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
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 usable. 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**

select Quick Info

The Windows Sockets **select** function determines the status of one or more sockets, waiting if necessary.

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

Parameters

nfds
[in] This argument is ignored and included only for the sake of compatibility.

readfds
[in/out] An optional pointer to a set of sockets to be checked for readability.

writefds
[in/out] An optional pointer to a set of sockets to be checked for writability

exceptfds
[in/out] An optional pointer to a set of sockets to be checked for errors.

timeout
[in] 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 can 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. 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** is guaranteed not to block.

For connection-oriented sockets, readability can also indicate that a close request has been received from the peer. If the virtual circuit was closed gracefully, then a **recv** will return immediately with zero 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 (nonblocking), 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** are guaranteed to succeed. However, they can 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 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 (nonblocking), failure of the connect attempt is indicated in *exceptfds*. This specification does not define which other errors will be included.

Any two of *readfds*, *writefds*, or *exceptfds* can 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/terminated

writefds:

- If **connect**ing (nonblocking), connection has succeeded
- Data can be sent

exceptfds:

- If **connect**ing (nonblocking), 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 can 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*. Otherwise, zero.

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** can 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 Windows Sockets will not yield.

Return Values

select returns the total number of descriptors which are ready and contained in the fd_set structures, zero if the time limit expired, or SOCKET_ERROR if an error occurred. If the return value is SOCKET_ERROR, **WSAGetLastError** can 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 function.
WSAEFAULT	The Windows Sockets implementation was unable to allocated 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	The (blocking) call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking Windows Sockets 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**

send Quick Info

The Windows Sockets **send** function sends data on a connected socket.

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

Parameters

s
[in] A descriptor identifying a connected socket.

buf
[in] A buffer containing the data to be transmitted.

len
[in] The length of the data in *buf*.

flags
[in] 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 nonblocking I/O mode. On nonblocking stream-oriented sockets, the number of bytes written can be between 1 and the requested length, depending on buffer availability on both the local and foreign hosts. The **select**, **WSAAsyncSelect** or **WSAEventSelect** call can be used to determine when it is possible to send more data.

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

Flags can 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 Windows Sockets service provider can choose to ignore this flag.
MSG_OOB	Send out-of-band data (stream-style socket such as SOCK_STREAM only. Also see Out-Of-Band data for a discussion of this topic).

Return Values

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set.
WSAEINTR	The (blocking) call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking Windows Sockets 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 the remote host resetting.
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 shut down; 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 nonblocking 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 cannot 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 terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
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 usable.
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**, **sendto**, **socket**, **WSAAsyncSelect**, **WSAEventSelect**

sendto Quick Info

The Windows Sockets **sendto** function sends data to a specific destination.

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

Parameters

s
[in] A descriptor identifying a socket.

buf
[in] A buffer containing the data to be transmitted.

len
[in] The length of the data in *buf*.

flags
[in] Specifies the way in which the call is made.

to
[in] An optional pointer to the address of the target socket.

tolen
[in] 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 can 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 can 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:

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 can 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 nonblocking I/O mode. On nonblocking stream-oriented sockets, the number of bytes written can be between 1 and the requested length, depending on buffer availability on both the local and foreign hosts. The **select**, **WSAAsyncSelect** or **WSAEventSelect** call can be used to determine when it is possible to send more data.

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

Flags can 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 Windows Sockets service provider can choose to ignore this flag.
MSG_OOB	Send out-of-band data (stream-style socket such as SOCK_STREAM only. Also see Out-Of-Band data for a discussion of this topic.)

Return Values

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set.
WSAEINVAL	An unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled.
WSAEINTR	The (blocking) call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking Windows Sockets 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 the remote host resetting.
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 shut down; 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 nonblocking 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 cannot be reached from this host at this time.
WSAECONNABORTED	The virtual circuit was terminated due to a time-out or other failure. The application should close the socket as it is no longer usable.
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 usable.
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAEDESTADDRREQ	A destination address is required.
WSAENETUNREACH	The network cannot 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**, **send**, **socket**, **WSAAsyncSelect**, **WSAEventSelect**

SetService Quick Info

Important The **SetService** function is obsolete. For the convenience of Windows Sockets 1.1 developers, the reference material is below.

In Windows Sockets 2, this functionality is realized with the functions detailed in *Protocol-Independent Name Resolution*.

The **SetService** function registers or deregisters a network service within

one or more name spaces. The function can also add or remove a network service type within one or more name spaces.

```
INT SetService(
  DWORD dwNameSpace,                    // specifies name space(s) to operate within
  DWORD dwOperation,                    // specifies operation to perform
  DWORD dwFlags,                        // set of bit flags that modify function operation
  LPSERVICE_INFO lpServiceInfo,         // points to structure containing service information
  LPSERVICE_ASYNC_INFO lpServiceAsyncInfo,  // reserved for future use, must be NULL
  LPDWORD lpdwStatusFlags               // points to set of status bit flags
);
```

Parameters

dwNameSpace

Specifies the name space, or a set of default name spaces, within which the function will operate.

Use one of the following constants to specify a name space:

Value	Name Space
NS_DEFAULT	A set of default name spaces. The function queries each name space within this set. The set of default name spaces typically includes all the name spaces installed on the system. System administrators, however, can exclude particular name spaces from the set. NS_DEFAULT is the value that most applications should use for *dwNameSpace*.
NS_DNS	The Domain Name System used in the Internet to resolve the name of the host.
NS_NDS	The NetWare 4 provider.
NS_NETBT	The NetBIOS over TCP/IP layer. All Windows NT and Windows 95 systems register their computer names with NetBIOS. This name space is used to convert a computer name to an IP address that uses this registration.
NS_SAP	The NetWare Service Advertising Protocol. This can access the Netware bindery, if appropriate. NS_SAP is a dynamic name space that enables the registration of services.
NS_TCPIP_HOSTS	Lookup value in the <systemroot>\system32\drivers\etc\posts file.

NS_TCPIP_LOCAL	Local TCP/IP name resolution mechanisms, including comparisons against the local host name and lookup value in the cache of host to IP address mappings.

dwOperation

Specifies the operation that the function will perform. Use one of the following values to specify an operation:

Value	Meaning
SERVICE_REGISTER	Register the network service with the name space. This operation may be used with the SERVICE_FLAG_DEFER and SERVICE_FLAG_HARD bit flags.
SERVICE_DEREGISTER	Deregister the network service from the name space. This operation can be used with the SERVICE_FLAG_DEFER and SERVICE_FLAG_HARD bit flags.
SERVICE_FLUSH	Perform any operation that was called with the SERVICE_FLAG_DEFER bit flag set to one.
SERVICE_ADD_TYPE	Add a service type to the name space. For this operation, use the **ServiceSpecificInfo** member of the **SERVICE_INFO** structure pointed to by *lpServiceInfo* to pass a **SERVICE_TYPE_INFO_ABS** structure. You must also set the **ServiceType** member of the **SERVICE_INFO** structure. Other **SERVICE_INFO** members are ignored.
SERVICE_DELETE_TYPE	Remove a service type, added by a previous call specifying the SERVICE_ADD_TYPE operation, from the name space.

dwFlags

A set of bit flags that modify the function's operation. You can set one or more of the following bit flags:

Value	Name Space
SERVICE_FLAG_DEFER	This bit flag is valid only if the operation is SERVICE_REGISTER or SERVICE_DEREGISTER. If this bit flag is one, and it is valid, the name-space provider should defer the registration or deregistration operation until a SERVICE_FLUSH operation is requested.
SERVICE_FLAG_HARD	This bit flag is valid only if the operation is SERVICE_REGISTER or SERVICE_DEREGISTER. If this bit flag is one, and it is valid, the name-space provider updates any

	relevant persistent store information when the operation is performed. For example: If the operation involves deregistration in a name space that uses a persistent store, the name-space provider would remove the relevant persistent store information.

lpService_Info
Points to a **SERVICE_INFO** structure that contains information about the network service or service type.

lpServiceAsyncInfo
This parameter is reserved for future use. It must be set to NULL.

lpdwStatusFlags
A set of bit flags that receive function status information. The following bit flag is defined:

Value	Meaning
SET_SERVICE_PARTIAL_SUCCESS	One or more name-space providers were unable to successfully perform the requested operation.

Return Values

If the function succeeds, the return value is not SOCKET_ERROR.

If the function fails, the return value is SOCKET_ERROR. To get extended error information, call **GetLastError**. **GetLastError** may return the following extended error value:

Value	Meaning
ERROR_ALREADY_REGISTERED	The function tried to register a service that was already registered.

See Also

GetService, **SERVICE_INFO**, **SERVICE_TYPE_INFO_ABS**

setsockopt Quick Info

The Windows Sockets **setsockopt** function sets a socket option.

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

Parameters

s
[in] A descriptor identifying a socket.

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

optname
[in] The socket option for which the value is to be set.

optval
[in] A pointer to the buffer in which the value for the requested option is supplied.

optlen
[in] 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 can 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 can be sent on the socket.

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**. 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	Do not 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	Do not route: send directly to interface.

SO_GROUP_PRIORITY	int	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 Out-Of-Band data for a discussion of this topic.)
SO_RCVBUF	int	Specify buffer size for receives
SO_REUSEADDR	BOOL	Allow the socket to be bound to an address which is already in use. (See **bind**.)
SO_SNDBUF	int	Specify buffer size for sends.
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**[1]

TCP_NODELAY	BOOL	Disables the Nagle algorithm for send coalescing.

1 included for backward compatibility with Windows Sockets 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 time-out
SO_SNDLOWAT	int	Send low water mark
SO_SNDTIMEO	int	Send time-out
SO_TYPE	int	Type of the socket

SO_DEBUG

Windows Sockets 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

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 nongroup sockets or for service providers which do not support group sockets.

SO_KEEPALIVE

An application can 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 Windows Sockets 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 nonzero value, set *l_linger* to zero or the desired time-out (in seconds), and call **setsockopt**. To enable SO_DONTLINGER (that is, disable SO_LINGER) *l_onoff* should be set to zero and **setsockopt** should be called. Note that enabling SO_LINGER with a nonzero time-out on a nonblocking socket is not recommended.

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

SO_REUSEADDR

By default, a socket cannot be bound (see **bind**) to a local address which is already in use. On occasion, however, it may be necessary 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 Windows Sockets 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 can request different buffer sizes (larger or smaller). The call to **setsockopt** can 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 can 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 Values

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
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 Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	*level* is not valid, or the information in *optval* is not valid.
WSAENETRESET	Connection has timed out when SO_KEEPALIVE is set.
WSAENOPROTOOPT	The option is unknown or unsupported for the specified provider.
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**

shutdown Quick Info

The Windows Sockets **shutdown** function disables sends and/or receives on a socket.

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

Parameters

s

[in] A descriptor identifying a socket.

how

[in] 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.

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:

1. call **WSAAsyncSelect** to register for FD_CLOSE notification,
2. call **shutdown** with *how*=SD_SEND,
3. when FD_CLOSE received, call **recv** until zero returned, or SOCKET_ERROR, and
4. 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 Windows Sockets provider is not required to support the use of **connect** on such a socket.

Return Values

If no error occurs, **shutdown** returns zero. Otherwise, a value of SOCKET_ERROR is returned, and a specific error code can be retrieved by calling **WSAGetLastError**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.

WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	*how* is not valid, or is not consistent with the socket type, for example, SD_SEND is used with a UNI_RECV socket type.
WSAEINPROGRESS	A blocking Windows Sockets 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**

socket Quick Info

The Windows Sockets **socket** function creates a socket which is bound to a specific service provider.

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

Parameters

af
[in] An address family specification.

type
[in] A type specification for the new socket.

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

Remarks

The **socket** function causes a socket descriptor and any related resources to be allocated and bound to a specific transport service provider. Windows Sockets will utilize the first available service provider that supports the requested combination of address family, socket type and protocol parameters. Note that the socket created will have the overlapped attribute. Sockets without the overlapped attribute can only be created by using WSASocket.

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

The following are the only two *type* specifications supported for Windows Sockets 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 Windows Sockets 2, many new socket types will be introduced. However, since an application can dynamically discover the attributes of each available transport protocol through 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 can be sent or received on it. A connection to another socket is created with a **connect** call. Once connected, data can 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 can be sent to that peer using **send** and can be received only from this peer using **recv**.

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

Return Values

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 can be retrieved by calling **WSAGetLastError**.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
WSAENETDOWN	The network subsystem or the associated service provider has failed.
WSAEAFNOSUPPORT	The specified address family is not supported.
WSAEINPROGRESS	A blocking Windows Sockets 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**, **ioctlsocket**, **listen**, **recv**, **recvfrom**, **select**, **send**, **sendto**, **setsockopt**, **shutdown**, **WSASocket**

TransmitFile Quick Info

Notice This function is a Microsoft-specific extension to the Windows Sockets specification. For more information, see Microsoft Extensions and Windows Sockets 2.

The Windows Sockets **TransmitFile** function transmits file data over a connected socket handle. This function uses the operating system's cache manager to retrieve the file data, and provides high-performance file data transfer over sockets.

```
BOOL TransmitFile(
  SOCKET hSocket,
  HANDLE hFile,
  DWORD nNumberOfBytesToWrite,
  DWORD nNumberOfBytesPerSend,
  LPOVERLAPPED lpOverlapped,
  LPTRANSMIT_FILE_BUFFERS lpTransmitBuffers,
  DWORD dwFlags
);
```

Parameters

hSocket

A handle to a connected socket. The function will transmit the file data over this socket.

The socket specified by *hSocket* must be a connection-oriented socket.

Sockets of type SOCK_STREAM, SOCK_SEQPACKET, or SOCK_RDM are connection-oriented sockets. The **TransmitFile** function does not support datagram sockets.

hFile

A handle to an open file. The function transmits this file's data. The operating system reads the file data sequentially. You can improve caching performance by opening the handle with the FILE_FLAG_SEQUENTIAL_SCAN.

nNumberOfBytesToWrite

The number of bytes to transmit. The function completes when it has sent this many bytes, or if an error occurs.

Set this parameter to zero to transmit the entire file.

nNumberOfBytesPerSend

The size of each block of data sent per send operation. This specification is for use by the sockets layer of the operating system.

Set this parameter to zero to have the sockets layer select a default send size.

This parameter is useful for message protocols that have limitations on the size of individual send requests.

lpOverlapped

Pointer to an **OVERLAPPED** structure. If the socket handle has been opened as overlapped, specify this parameter in order to achieve an overlapped (aysnchronous) I/O operation. By default, socket handles are opened as overlapped.

You can use *lpOverlapped* to specify an offset within the file at which to start the file data transfer by setting the **Offset** and **OffsetHigh** member of the **OVERLAPPED** structure. If *lpOverlapped* is NULL, the transmission of data always starts at the current byte offset in the file.

When *lpOverlapped* is not NULL, the overlapped I/O might not finish before **TransmitFile** returns. In that case, the **TransmitFile** function returns FALSE, and **GetLastError** returns ERROR_IO_PENDING. This lets the caller continue processing while the file transmission operation completes. The operating system will set the event specified by the **hEvent** member of the **OVERLAPPED** structure, or the socket specified by *hSocket*, to the signaled state upon completion of the data transmission request.

lpTransmitBuffers

Pointer to a **TRANSMIT_FILE_BUFFERS** data structure that contains pointers to data to send before and after the file data is sent. Set this parameter to NULL if you only want to transmit the file data.

dwFlags

An attribute that has three settings:

TF_DISCONNECT

Start a transport-level disconnect after all the file data has been queued for transmission.

TF_REUSE_SOCKET

Prepare the socket handle to be reused. When the TransmitFile request completes, the socket handle can be passed to the **AcceptEx** function. It is only valid if TF_DISCONNECT is also specified.

TF_WRITE_BEHIND

Complete the **TransmitFile** request immediately, without pending. If this flag is specified and **TransmitFile** succeeds, then the data has been accepted by the system but not necessarily acknowledged by the remote end. If **TransmitFile** returns TRUE, there will be no completion port indication for the I/O. Do not use this setting with the other two settings.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. To get extended error information, call **GetLastError**. The function returns FALSE if an overlapped I/O operation is not complete before **TransmitFile** returns. In that case, **GetLastError** returns ERROR_IO_PENDING.

Remarks

The Windows NT Server optimizes the **TransmitFile** function for high performance. The Windows NT Workstation optimizes the function for minimum memory and resource utilization. Expect better performance results when using **TransmitFile** on Windows NT Server.

WSAAccept Quick Info

Overview

Group

The Windows Sockets **WSAAccept** function conditionally accepts a connection based on the return value of a condition function, and optionally creates and/or joins a socket group.

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

Parameters

s

[in] A descriptor identifying a socket which is listening for connections after a **listen**.

addr

[out] 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

[in/out] An optional pointer to an integer which contains the length of the address *addr*.

lpfnCondition

[in] 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

[in] The callback data passed back to the application as a condition function parameter. This parameter is not interpreted by Windows Sockets.

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 (that is, 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 nonblocking mode, if this function is called when no pending connections are present on the queue, **WSAAccept** returns an error as described below. The accepted socket cannot be used to accept more connections. The original socket remains open.

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. In 16-bit Windows environments, it is invoked in the same thread as **WSAAccept**, thus no other Windows Sockets functions can be called except **WSAIsBlocking** and **WSACancelBlockingCall**. 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.

lpSQOS references the flow specifications for socket *s* specified by the caller, one for each direction, followed by any additional provider-specific parameters. The sending or receiving flow specification values will be ignored as appropriate for any unidirectional sockets. A NULL value for *lpSQOS* indicates no caller supplied QOS. QOS information can be returned if a QOS negotiation is to occur.

lpGQOS references the flow specifications 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 can 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 *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.

The result parameter *g* is assigned within the condition function to indicate the following actions:

1. if &*g* is an existing socket group ID, add *s* to this group, provided all the requirements set by this group are met; or
2. if &*g* = SG_UNCONSTRAINED_GROUP, create an unconstrained socket group and have *s* as the first member; or

3. if &*g* = SG_CONSTRAINED_GROUP, create a constrained socket group and have *s* as the first member; or
4. if &*g* = zero, no group operation is performed.

For unconstrained groups, any set of sockets can be grouped together as long as they are supported by a single service provider. A constrained socket group can 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.

Return Values

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 can 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 function.
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 the *lpfnCondition* is not part of the user address space.
WSAEINTR	The (blocking) call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking Windows Sockets 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 nonempty 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 nonblocking 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**

WSAAddressToString Quick Info

The Windows Sockets **WSAAddressToString** function converts all components of a SOCKADDR structure into a human-readable string representation of the address.

This is intended to be used mainly for display purposes. If the caller wants the translation to be done by a particular provider, it should supply the corresponding WSAPROTOCOL_INFO structure in the *lpProtocolInfo* parameter.

```
INT WSAAddressToString(
  LPSOCKADDR lpsaAddress,
  DWORD dwAddressLength,
  LPWSAPROTOCOL_INFO lpProtocolInfo,
  OUT LPTSTR lpszAddressString,
  OUT LPDWORD lpdwAddressStringLength
);
```

Parameters

lpsaAddress
[in] Points to a **SOCKADDR** structure to translate into a string.

dwAddressLength
[in] The length of the Address **SOCKADDR**.

lpProtocolInfo
[in] (Optional) The WSAPROTOCOL_INFO structure for a particular provider.

lpszAddressString
[in] A buffer which receives the human-readable address string.

lpdwAddressStringLength
[in] On input, the length of the AddressString buffer. On output, returns the length ofthe string actually copied into the buffer.

Return Values

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

Error Codes

WSAEFAULT	The specified AddressString buffer is too small. Pass in a larger buffer
WSAEINVAL	The specified address is not a valid socket address.

WSAAsyncGetHostByAddr Quick Info

The Windows Sockets **WSAAsyncGetHostByAddr** function gets host information corresponding to an address−asynchronous version.

```
HANDLE WSAAsyncGetHostByAddr (
  HWND hWnd,
  unsigned int wMsg,
  const char FAR * addr,
  int len,
  int type,
  char FAR * buf,
  int buflen
);
```

Parameters

hWnd
[in] The handle of the window which should receive a message when the asynchronous request completes.

wMsg
[in] The message to be received when the asynchronous request completes.

addr
[in] A pointer to the network address for the host. Host addresses are stored in network byte order.

len
[in] The length of the address.

type
[in] The type of the address.

buf
[out] 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 Windows Sockets 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
[in] 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. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque "asynchronous task handle" which the application can 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 can 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 can reissue the **WSAAsyncGetHostByAddr** function call

with a buffer large enough to receive all the desired information (that is, 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 Values

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 can be retrieved by calling **WSAGetLastError**.

Comments

The buffer supplied to this function is used by Windows Sockets to construct a 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 can be set when an application window receives a message. As described above, they can 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	Nonrecoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can 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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 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 Windows Sockets implementation.

See Also

gethostbyaddr, **hostent**, **WSACancelAsyncRequest**

WSAAsyncGetHostByName Quick Info

The Windows Sockets **WSAAsyncGetHostByName** function gets host information corresponding to a hostname−asynchronous version.

```
HANDLE WSAAsyncGetHostByName (
  HWND hWnd,
  unsigned int wMsg,
  const char FAR * name,
  char FAR * buf,
  int buflen
);
```

Parameters

hWnd
[in] The handle of the window which should receive a message when the asynchronous request completes.

wMsg
[in] The message to be received when the asynchronous request completes.

name
[in] A pointer to the null terminated name of the host.

buf
[out] 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 Windows Sockets 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
[in] 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. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque "asynchronous task handle" which the application can 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 can 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 can reissue the **WSAAsyncGetHostByName** function call with a buffer large enough to receive all the desired information (that is, 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 Values

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 can be retrieved by calling **WSAGetLastError**.

Comments

The buffer supplied to this function is used by Windows Sockets 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 can be set when an application window receives a message. As described above, they can 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	Nonrecoverable errors, FORMERR, REFUSED, NOTIMP.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can 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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 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 Windows Sockets implementation.

See Also

gethostbyname, **hostent**, **WSACancelAsyncRequest**

WSAAsyncGetProtoByName Quick Info

The Windows Sockets **WSAAsyncGetProtoByName** function gets protocol information corresponding to a protocol name−asynchronous version.

```
HANDLE WSAAsyncGetProtoByName (
  HWND hWnd,
  unsigned int wMsg,
  const char FAR * name,
  char FAR * buf,
  int buflen
);
```

Parameters

hWnd
[in] The handle of the window which should receive a message when the asynchronous request completes.

wMsg
[in] The message to be received when the asynchronous request completes.

name
[in] A pointer to the null terminated protocol name to be resolved.

buf
[out] 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 Windows Sockets 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
[out] 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. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque "asynchronous task handle" which the application can 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 can 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 can reissue the **WSAAsyncGetProtoByName** function call with a buffer large enough to receive all the desired information (that is, 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 Values

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 can be retrieved by calling **WSAGetLastError**.

Comments

The buffer supplied to this function is used by Windows Sockets 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 can be set when an application window receives a message. As described above, they can 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	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can 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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 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 Windows Sockets implementation.

See Also

getprotobyname, **WSACancelAsyncRequest**

WSAAsyncGetProtoByNumber Quick Info

The Windows Sockets **WSAAsyncGetProtoByNumber** function gets protocol information corresponding to a protocol number—asynchronous version.

```
HANDLE WSAAsyncGetProtoByNumber (
  HWND hWnd,
  unsigned int wMsg,
  int number,
  char FAR * buf,
  int buflen
);
```

Parameters

hWnd
[in] The handle of the window which should receive a message when the asynchronous request completes.

wMsg
[in] The message to be received when the asynchronous request completes.

number
[in] The protocol number to be resolved, in host byte order.

buf
[out] 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 Windows Sockets 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
[in] 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. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque "asynchronous task handle" which the application can 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 can 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 can reissue the **WSAAsyncGetProtoByNumber** function call with a buffer large enough to receive all the desired information (that is, 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 Values

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 can be retrieved by calling **WSAGetLastError**.

Comments

The buffer supplied to this function is used by Windows Sockets 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 can be set when an application window receives a message. As described above, they can 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	Nonrecoverable errors, the protocols database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can 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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 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 Windows Sockets implementation.

See Also

getprotobynumber, **WSACancelAsyncRequest**

WSAAsyncGetServByName Quick Info

The Windows Sockets **WSAAsyncGetServByName** function gets service information corresponding to a service name and port−asynchronous version.

```
HANDLE WSAAsyncGetServByName (
  HWND hWnd,
  unsigned int wMsg,
  const char FAR * name,
  const char FAR * proto,
  char FAR * buf,
  int buflen
);
```

Parameters

hWnd

[in] The handle of the window which should receive a message when the asynchronous request completes.

wMsg

[in] The message to be received when the asynchronous request completes.

name

[in] A pointer to a null terminated service name.

proto

[in] A pointer to a protocol name. This can 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

[out] 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 Windows Sockets 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

[in] 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. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque "asynchronous task handle" which the application can 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 can 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 can reissue the **WSAAsyncGetServByName** function call with a buffer large enough to receive all the desired information (that is, 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 Values

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 can be retrieved by calling **WSAGetLastError**.

Comments

The buffer supplied to this function is used by Windows Sockets 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 can be set when an application window receives a message. As described above, they can 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	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can 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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 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 Windows Sockets implementation.

See Also

getservbyname, **WSACancelAsyncRequest**

WSAAsyncGetServByPort Quick Info

The Windows Sockets **WSAAsyncGetServByPort** function gets service information corresponding to a port and protocol−asynchronous version.

```
HANDLE WSAAsyncGetServByPort (
  HWND hWnd,
  unsigned int wMsg,
  int port,
  const char FAR * proto,
  char FAR * buf,
  int buflen
);
```

Parameters

hWnd

[in] The handle of the window which should receive a message when the asynchronous request completes.

wMsg

[in] The message to be received when the asynchronous request completes.

port

[in] The port for the service, in network byte order.

proto

[in] A pointer to a protocol name. This can 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

[out] 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 Windows Sockets 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

[in] 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. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque "asynchronous task handle" which the application can 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 can 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 can reissue the **WSAAsyncGetServByPort** function call with a buffer large enough to receive all the desired information (that is, 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 Values

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 can be retrieved by calling **WSAGetLastError**.

Comments

The buffer supplied to this function is used by Windows Sockets 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 can be set when an application window receives a message. As described above, they can 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	Nonrecoverable errors, the services database is not accessible.
WSANO_DATA	Valid name, no data record of requested type.

The following errors can 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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 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 Windows Sockets implementation.

See Also

getservbyport, **WSACancelAsyncRequest**

WSAAsyncSelect Quick Info

The Windows Sockets **WSAAsyncSelect** function requests Windows message-based notification of network events for a socket.

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

Parameters

s
[in] A descriptor identifying the socket for which event notification is required.

hWnd
[in] A handle identifying the window which should receive a message when a network event occurs.

wMsg
[in] The message to be received when a network event occurs.

lEvent
[in] A bitmask which specifies a combination of network events in which the application is interested.

Remarks

This function is used to request that the Windows Sockets 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 nonblocking mode, regardless of the value of *lEvent*. See **ioctlsocket** about how to set the nonoverlapped 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
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	Want to receive notification of socket group Quality of Service (QOS) changes

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 (that is, to indicate that Windows Sockets 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 can 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.

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

The error and event codes can 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 can 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 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	Quality of Service associated with the socket group to which *s* belongs has changed

Return Values

The return value is zero 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 can 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 can receive a message and issue a Windows Sockets 2 call which returns WSAEWOULDBLOCK immediately. For example, the following sequence of events is possible:

1. data arrives on socket **s**; Windows Sockets 2 posts **WSAAsyncSelect** message
2. application processes some other message
3. while processing, application issues an **ioctlsocket(s, FIONREAD...)** and notices that there is data ready to be read
4. application issues a **recv(s,...)** to read the data
5. application loops to process next message, eventually reaching the **WSAAsyncSelect** message indicating that data is ready to read
6. application issues **recv(s,...)**, which fails with the error WSAEWOULDBLOCK.

Other sequences are possible.

The Windows Sockets 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 re-enables 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	**WSAIoctl** with command SIO_GET_GROUP_QOS

Any call to the re-enabling routine, even one which fails, results in re-enabling 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 re-enabling 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:

1. Network transport stack receives 100 bytes of data on socket **s** and causes Windows Sockets 2 to post an FD_READ message.
2. The application issues **recv(s, buffptr, 50, 0)** to read 50 bytes.
3. 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 can receive multiple FD_READ messages. Such an application may need 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.

If any event has already happened when the application calls **WSAAsyncSelect** or when the re-enabling 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, Windows Sockets 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 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 zero, 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 specification 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.

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 datagram socket.
- **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)
- **FD_CLOSE**: Only valid on connection-oriented sockets (for example, 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 (for example, TCP FIN), when no data currently available to receive,
  4. when remote system terminates connection (for example, 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**:
  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.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
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 Windows Sockets 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
WSAEADDRINUSE	The specified address is already in use.
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAENETUNREACH	The network cannot 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 terminated due to a time-out or other failure.

Event: FD_READ

Event: FD_WRITE

Event: FD_OOB

Event: FD_ACCEPT

Event: FD_QOS

Event: FD_GROUP_QOS

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.

See Also

select, **WSAEventSelect**

WSACancelAsyncRequest Quick Info

The Windows Sockets **WSACancelAsyncRequest** function cancels an incomplete asynchronous operation.

```
int WSACancelAsyncRequest (
  HANDLE hAsyncTaskHandle
);
```

Parameters

hAsyncTaskHandle
[in] 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 Values

The value returned by **WSACancelAsyncRequest** is zero 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: zero is always an invalid asynchronous task handle.] The Windows Sockets specification does not prescribe how a conformant Windows Sockets provider should distinguish between the two cases. For maximum portability, a Windows Sockets application should treat the two errors as equivalent.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that the specified asynchronous task handle was invalid
WSAEINPROGRESS	A blocking Windows Sockets 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**, **WSAAsyncGetProtoByName**,

WSAAsyncGetProtoByNumber, **WSAAsyncGetServByName**, **WSAAsyncGetServByPort**

WSACancelBlockingCall Quick Info

This function has been removed in compliance with the Windows Sockets 2 specification, revision 2.2.0.

The function is not exported directly by the WS2_32.DLL, and Windows Sockets 2 applications should not use this function. Windows Sockets 1.1 applications that call this function are still supported through the WINSOCK.DLL and WSOCK32.DLL.

Blocking hooks are generally used to keep a single-threaded GUI application responsive during calls to blocking functions. Instead of using blocking hooks, an applications should use a separate thread (separate from the main GUI thread) for network activity.

WSACleanup Quick Info

The Windows Sockets **WSACleanup** function terminates use of the Windows Sockets DLL.

int WSACleanup (void);

Remarks

An application or DLL is required to perform a successful **WSAStartup** call before it can use Windows Sockets services. When it has completed the use of Windows Sockets, the application or DLL must call **WSACleanup** to deregister itself from a Windows Sockets 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 Windows Sockets 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 Windows Sockets DLL.

Return Values

The return value is zero 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 Windows Sockets programming error. 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 Windows Sockets operations for all threads.

Error Codes

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

See Also

closesocket, **shutdown**, **WSAStartup**

WSACloseEvent Quick Info

Overview

Group

The Windows Sockets **WSACloseEvent** function closes an open event object handle.

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

Parameters

hEvent

[in] 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 Values

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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 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**

WSAConnect Quick Info

Overview

Group

The Windows Sockets **WSAConnect** function establishes a connection to a peer, exchanges connect data, and specifies needed quality of service based on the supplied flow specification.

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

Parameters

s
[in] A descriptor identifying an unconnected socket.

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

namelen
[in] The length of the *name*.

lpCallerData
[in] A pointer to the user data that is to be transferred to the peer during connection establishment.

lpCalleeData
[out] A pointer to the user data that is to be transferred back from the peer during connection establishment.

lpSQOS
[in] A pointer to the flow specifications for socket *s*, one for each direction.

lpGQOS
[in] A pointer to the flow specifications 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 (for example, 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 reconnect an active connection will fail with the error code WSAEISCONN.

For a connectionless socket (for example, type SOCK_DGRAM), the operation erformed 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." Then, the default remote address will be indeterminate, so **send/WSASend** and **recv/WSARecv** calls will return the error code WSAENOTCONN. However, **sendto/WSASendTo** and **recvfrom/WSARecvFrom** can 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 zero 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 nonblocking 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 through **WSAIoctl** with either the SIO_SET_QOS or SIO_SET_GROUP_QOS opcodes.

lpSQOS specifies the flow specifications 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 specification 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 zero, respectively. A NULL value for *lpSQOS* indicates no application supplied QOS.

lpGQOS specifies the flow specifications 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 *lpSQOS->ProviderSpecific* should be set to NULL and zero, 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 (that is, 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 Values

If no error occurs, **WSAConnect** returns zero. 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 nonblocking 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, 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.

For a nonblocking socket, until the connection attempt completes all subsequent calls to **WSAConnect** on the same socket will fail with the error code WSAEALREADY.

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

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The specified address is already in use.
WSAEINTR	The (blocking) call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	A nonblocking **connect/WSAConnect** call is in progress on the specified socket.
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
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 cannot 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 specifications specified in *lpSQOS* and *lpGQOS* cannot be satisfied.
WSAEPROTONOSUPPORT	The *lpCallerData* augment is not supported by the service provider.
WSAETIMEDOUT	Attempt to connect timed out without establishing a connection.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the connection cannot be completed immediately. It is possible to **select** the socket while it is connecting by **select**ing it for writing.
WSAEACCES	Attempt to connect datagram socket to broadcast address failed because **setsockopt** SO_BROADCAST is not enabled.

See Also

accept, **bind**, **connect**, **getsockname**, **getsockopt**, **select**, **socket**, **WSAAsyncSelect**, **WSAEventSelect**

WSACreateEvent Quick Info

Overview

Group

The Windows Sockets **connect** function creates a new event object.

WSAEVENT WSACreateEvent(void);

Remarks

The event object created by this function is manual reset, with an initial state of nonsignaled. Windows Sockets 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 function directly. The scope of an event object is limited to the process in which it is created.

Return Values

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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 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**

WSADuplicateSocket Quick Info

Quick Info

Quick Info

The Windows Sockets **WSADuplicateSocket** function returns a WSAPROTOCOL_INFO structure that can be used to create a new socket descriptor for a shared socket.

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

Parameters

s
[in] Specifies the local socket descriptor.

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

lpProtocolInfo
[out] A pointer to a buffer allocated by the client that is large enough to contain a WSAPROTOCOL_INFO structure. The service provider copies the protocol info structure 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.

Sockets can be shared among threads in a given process without using the **WSADuplicateSocket** function, since a socket descriptor is valid in all of a process's 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 Values

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 Windows Sockets 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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAEINPROGRESS	A blocking Windows Sockets 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.

See Also

WSASocket

WSAEnumNameSpaceProviders Quick Info

The Windows Sockets **WSAEnumNameSpaceProviders** function retrieves information about available name spaces.

```
INT WSAAPI WSAEnumNameSpaceProviders (
  LPDWORD lpdwBufferLength,
  LPWSANAMESPACE_INFO lpnspBuffer
);
```

Parameters

lpdwBufferLength

[in/out] On input, the number of bytes contained in the buffer pointed to by *lpnspBuffer*. On output (if the function 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

[out] A buffer which is filled with **WSANAMESPACE_INFO** structuresdescribed 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**.

Return Values

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**.

Error Codes

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 Windows Sockets 2 DLL has not been initialized. The application must first call **WSAStartup** before calling any Windows Sockets functions.

WSAEnumNetworkEvents Quick Info

Quick Info

Quick Info

The Windows Sockets **WSAEnumNetworkEvents** function discovers occurrences of network events for the indicated socket.

```
int WSAEnumNetworkEvents (
  SOCKET s,
  WSAEVENT hEventObject,
  LPWSANETWORKEVENTS lpNetworkEvents
);
```

Parameters

s

[in] A descriptor identifying the socket.

hEventObject

[in] An optional handle identifying an associated event object to be reset.

lpNetworkEvents

[out] A pointer to a WSANETWORKEVENTS structure 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 nonzero *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**.

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 Windows Sockets 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 iErrorCodes[FD_MAX_EVENTS];
} WSANETWORKEVENTS, FAR * LPWSANETWORKEVENTS;
```

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

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

Event: FD_CONNECT

Error Code	Meaning
WSAEADDRINUSE	The specified address is already in use.
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAENETUNREACH	The network cannot 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 terminated due to a time-out or other failure.

Event: FD_READ

Event: FD_WRITE

Event: FD_OOB

Event: FD_ACCEPT

Event: FD_QOS

Event: FD_GROUP_QOS

Error Code	Meaning
WSAENETDOWN	The network subsystem has failed.

Return Values

The return value is zero 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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	Indicates that one of the specified parameters was invalid.
WSAEINPROGRESS	A blocking Windows Sockets 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

WSAEventSelect

WSAEnumProtocols Quick Info

Quick Info

Quick Info

The Windows Sockets **WSAEnumProtocols** function retrieves information about available transport protocols.

```
int WSAEnumProtocols (
  LPINT lpiProtocols,
  LPWSAPROTOCOL_INFO lpProtocolBuffer,
  ILPDWORD lpdwBufferLength
);
```

Parameters

lpiProtocols

[in] 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

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

lpdwBufferLength

[in/out] 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 structure 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 structures appear in the buffer coincides with the order in which the protocol entries were registered by the service provider with the Windows Sockets DLL, or with any subsequent re-ordering that may have occurred through the Windows Sockets applet supplied for establishing default TCP/IP providers.

Return Values

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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 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*.

WSAEventSelect Quick Info

Quick Info

Quick Info

The Windows Sockets **WSAEventSelect** function specifies an event object to be associated with the supplied set of FD_XXX network events.

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

Parameters

s
[in] A descriptor identifying the socket.

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

lNetworkEvents
[in] 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.

This function automatically sets socket *s* to nonblocking 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
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	Want to receive notification of socket group Quality of Service (QOS) changes

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.

Return Values

The return value is zero 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 Windows Sockets call which returns WSAEWOULDBLOCK immediately. For example, the following sequence of operations is possible:

1. data arrives on socket **s**; Windows Sockets sets the **WSAEventSelect** event object
2. application does some other processing
3. while processing, application issues an **ioctlsocket(s, FIONREAD...)** and notices that there is data ready to be read
4. application issues a **recv(s,...)** to read the data
5. application eventually waits on event object specified in **WSAEventSelect**, which returns immediately

indicating that data is ready to read

6. 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 re-enables 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
FD_GROUP_QOS	**WSAIoctl** with command SIO_GET_GROUP_QOS

Any call to the re-enabling routine, even one which fails, results in re-enabling 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 re-enabling 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:

1. Transport provider receives 100 bytes of data on socket **s** and causes Windows Sockets DLL to record the FD_READ network event and set the associated event object.
2. The application issues **recv(s, buffptr, 50, 0)** to read 50 bytes.
3. The transport provider causes Windows Sockets 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.

If a network event has already happened when the application calls **WSAEventSelect** or when the re-enabling 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, Windows Sockets 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 zero, 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 FIN WAIT or CLOSE WAIT states. This results from the remote end performing a **shutdown** on the send side or a **closesocket**.

Please note Windows Sockets 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 specification 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.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
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 Windows Sockets 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

WSACloseEvent, **WSACreateEvent**, **WSAEnumNetworkEvents**, **WSAWaitForMultipleEvents**

WSAGetLastError Quick Info

The Windows Sockets **WSAGetLastError** function gets the error status for the last operation which failed.

int WSAGetLastError (void);

Remarks

This function returns the last network error that occurred. When a particular Windows Sockets 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.

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.

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 Values

The return value indicates the error code for this thread's last Windows Sockets operation that failed.

See Also

getsockopt, **WSASetLastError**

WSAGetOverlappedResult Quick Info

Quick Info

Quick Info

The Windows Sockets **WSAGetOverlappedResult** function returns the results of an overlapped operation on the specified socket.

```
BOOL WSAGetOverlappedResult (
  SOCKET s,
  LPWSAOVERLAPPED lpOverlapped,
  LPDWORD lpcbTransfer,
  BOOL fWait,
  LPDWORD lpdwFlags
);
```

Parameters

s

[in] 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**.

lp*Overlapped*

[in] Points to a WSAOVERLAPPED structure that was specified when the overlapped operation was started.

pc*bTransfer*

[out] 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

[in] 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.

lpdwFlags

[out] Points to a 32-bit variable that will receive one or more flags that supplement the completion status. If the overlapped operation was initiated through **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 FALSE, 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.

Return Values

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 function.
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.

See Also

WSAAccept, **WSAConnect**, **WSACreateEvent**, **WSAIoctl**, **WSARecv**, **WSARecvFrom**, **WSASend**, **WSASendTo**, **WSAWaitForMultipleEvents**

WSAGetQOSByName Quick Info

Quick Info

Quick Info

The Windows Sockets **WSAGetQOSByName** function initializes a QOS structure based on a named template.

```
BOOL WSAGetQOSByName(
  SOCKET s,
  LPWSABUF lpQOSName,
  LPQOS lpQOS
);
```

Parameters

s
[in] A descriptor identifying a socket.

lpQOSName
[in] Specifies the QOS template name.

lpQOS
[out] A pointer to the QOS structure to be filled.

Remarks

Applications may use this function to initalize 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.

Return Values

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 function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The *lpQOS* argument is not a valid part of the user address space, or the buffer length for *lpQOS* is too small.
WSA_INVAL	The specified QOS template name is invalid.

See Also

getsockopt, **WSAAccept**, **WSAConnect**

WSAGetServiceClassInfo Quick Info

The Windows Sockets **WSAGetServiceClassInfo** function retrieves all of the class information (schema) pertaining to a specified service class from a specified name space provider.

```
INT WSAGetServiceClassInfo(
  LPGUID lpProviderId,
  LPGUID lpServiceClassId,
  LPDWORD lpdwBufSize,
  LPWSASERVICECLASSINFO lpServiceClassInfo
);
```

Parameters

lpProviderId
[in] Pointer to a GUID which identifies a specific name space provider

lpServiceClassId
[in] Pointer to a GUID identifying the service class in question

lpdwBufferLength
[in/out] On input, the number of bytes contained in the buffer pointed to by *lpServiceClassInfos*. On output, if the function fails and the error is WSAEFAULT, then it contains the minimum number of bytes to pass for the *lpServiceClassInfo* to retrieve the record.

lpServiceClassInfo
[out] 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 Service Class Data Structures for more information.

Return Values

The return value is zero 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

WSAEFAULT	The buffer referenced by *lpServiceClassInfo* is too small. Pass in a larger buffer.
WSA_INVALID_PARAMETER	the specified service class ID or name space provider ID is invalid.
WSANOTINITIALIZED	The Windows Sockets 2 DLL has not been initialized. The application must first call **WSAStartup** before calling any Windows Sockets functions.

WSAGetServiceClassNameByServiceClassId Quick Info

The Windows Sockets **WSAGetServiceClassNameByServiceClassId** function returns 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 WSAGetServiceClassNameByServiceClassId(
  LPGUID lpServiceClassId,
  LPTSTR lpszServiceClassName,
  LPDWORD lpdwBufferLength
);
```

Parameters

lpServiceClassId
[in] Pointer to the GUID for the service class.

lpszServiceClassName
[out] Sevice name.

lpdwBufferLength
[in/out] On input length of buffer returned by *lpszServiceClassName*. On output, the length of the service name copied into *lpszServiceClassName*.

Return Values

The return value is zero 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

WSAEFAULT	The specified ServiceClassName buffer is too small. Pass in a larger buffer
WSA_INVALID_PARAMETER	the specified ServiceClassId is invalid.
WSANOTINITIALIZED	The Windows Sockets 2 DLL has not been initialized. The application must first call **WSAStartup** before calling any Windows Sockets functions.

WSAHtonl Quick Info

Quick Info

Quick Info

The Windows Sockets **WSAHtonl** function converts a **u_long** from host byte order to network byte order.

```
int WSAHtonl (
  SOCKET s,
  u_long hostlong,
  u_long FAR * lpnetlong
);
```

Parameters

s
[in] A descriptor identifying a socket.

hostlong
[in] A 32-bit number in host byte order.

lpnetlong
[out] 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 Values

If no error occurs, **WSAHtonl** returns zero. 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 function.
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**, **ntohl**, **ntohs**, **WSANtohl**, **WSAHtons**, **WSANtohs**

WSAHtons Quick Info

Quick Info

Quick Info

The Windows Sockets **WSAHtons** function converts a **u_short** from host byte order to network byte order.

```
int WSAHtons (
  SOCKET s,
  u_short hostshort,
  u_short FAR * lpnetshort
);
```

Parameters

s
[in] A descriptor identifying a socket.

hostshort
[in] A 16-bit number in host byte order.

lpnetshort
[out] 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 Values

If no error occurs, **WSAHtons** returns zero. 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 function.
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**, **ntohl**, **ntohs**, **WSAHtonl**, **WSANtohl**, **WSANtohs**

WSAInstallServiceClass Quick Info

The Windows Sockets **WSAInstallServiceClass** function registers 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 WSAInstallServiceClass(
  LPWSASERVICECLASSINFO lpServiceClassInfo
);
```

Parameters

lpServiceClassInfo

[in] Service class to name space specific type mapping information. Multiple mappings can be handled at one time.

See section Service Class Data Structures for a description of pertinent data structures.

Return Values

The return value is zero 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

WSAEACCESS	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.
WSANOTINITIALIZED	The Windows Sockets 2 DLL has not been initialized. The application must first call **WSAStartup** before calling any Windows Sockets functions.

WSAIoctl Quick Info

Quick Info

Quick Info

The Windows Sockets **WSAIoctl** function controls the mode of a socket.

```
int WSAIoctl (
  SOCKET s,
  DWORD dwIoControlCode,
  LPVOID lpvInBuffer,
  DWORD cbInBuffer,
  LPVOID lpvOUTBuffer,
  DWORD cbOUTBuffer,
  LPDWORD lpcbBytesReturned,
  LPWSAOVERLAPPED lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

Parameters

s
[in] Handle to a socket

dwIoControlCode
[in] Control code of operation to perform

lpvInBuffer
[in] Address of input buffer

cbInBuffer
[in] Size of input buffer

lpvOutBuffer
[out] Address of output buffer

cbOutBuffer
[in] Size of output buffer

lpcbBytesReturned
[out] Address of actual bytes of output

lpOverlapped
[in] Address of WSAOVERLAPPED structure

lpCompletionRoutine
[in] A pointer to the completion routine called when the operation has been completed.

Remarks

This routine is used to set or retrieve operating parameters associated with the socket, the transport protocol, or the communications subsystem. For nonoverlapped 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 nonblocking mode, this function may return WSAEWOULDBLOCK if the specified operation cannot be finished immediately. In this case, the application should change the socket to the blocking mode and reissue the request. 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 through **WSAGetOverlappedResult**. The *lpcbBytesReturned* parameter is ignored.

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	3	2	2 2	2 2 2 2 2 2 2 1 1 1 1	1 1 1 1 1 1
1	0	9	8 7	6 5 4 3 2 1 0 9 8 7 6	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 and FIONBIO.

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 - The specific ioctl code for the operation.

The following Unix commands are supported:

Parameters

FIONBIO

Enable or disable nonblocking mode on socket *s*. *lpvInBuffer* points at an **unsigned long**, which is nonzero if nonblocking mode is to be enabled and zero if it is to be disabled. When a socket is created, it operates in blocking mode (that is, nonblocking 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 zero, or disable **WSAEventSelect** by calling **WSAEventSelect** with the *lNetworkEvents* parameter equal to zero.

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 (for example, 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. If *s* is message oriented (for example, 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 (for example, 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 Windows Sockets 2 commands are supported:

Parameters

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 (for example, TH_NETDEV and TH_TAPI.), 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.

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 structure 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, T==1)
: Retrieve the QOS structure associated with the socket. The input buffer is optional. Some protocols (for example, 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 structure. The WSAENOPROTOOPT error code is indicated for service providers which do not support QOS.

SIO_GET_GROUP_QOS (opcode setting: O, I, T==1)
Retrieve the QOS structure associated with the socket group to which this socket belongs. The input buffer is optional. Some protocols (for example, 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 structure 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.

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.

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)
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 (for example, TH_NETDEV and TH_TAPI). 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.

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 through the *cbTransferred* parameter in the completion routine (if specified), or through 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.

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 Values

Upon successful completion, the **WSAIoctl** returns zero. 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* or *lpcbBytesReturned* 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. (For examle, the flow specifications 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 nonblocking and the requested operation would block.

See Also

getsockopt, **ioctlsocket**, **setsockopt**, **socket**, **WSASocket**

WSAIsBlocking Quick Info

This function has been removed in compliance with the Windows Sockets 2 specification, revision 2.2.0.

The function is not exported directly by the WS2_32.DLL, and Windows Sockets 2 applications should not use this function. Windows Sockets 1.1 applications that call this function are still supported through the WINSOCK.DLL and WSOCK32.DLL.

Blocking hooks are generally used to keep a single-threaded GUI application responsive during calls to blocking functions. Instead of using blocking hooks, an applications should use a separate thread (separate from the main GUI thread) for network activity.

WSAJoinLeaf Quick Info

Quick Info

Quick Info

The Windows Sockets **connect** function joins a leaf node into a multipoint session, exchanges connect data, and specifies needed quality of service based on the supplied flow specifications.

```
SOCKET WSAJoinLeaf (
  SOCKET s,
  const struct sockaddr FAR * name,
  int namelen,
  LPWSABUF lpCallerData,
  LPWSABUF lpCalleeData,
  LPQOS lpSQOS,
  LPQOS lpGQOS,
  DWORD dwFlags
);
```

Parameters

s
: [in] A descriptor identifying a multipoint socket.

name
: [in] The name of the peer to which the socket is to be joined.

namelen
: [in] The length of the *name*.

lpCallerData
: [in] A pointer to the user data that is to be transferred to the peer during multipoint session establishment.

lpCalleeData
: [out] A pointer to the user data that is to be transferred back from the peer during multipoint session establishment.

lpSQOS
: [in] A pointer to the flow specifications for socket *s*, one for each direction.

lpGQOS
: [in] A pointer to the flow specifications for the socket group (if applicable).

dwFlags
: [in] 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. If the socket is in the nonblocking mode, the returned socket descriptor will not be usable 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. Refer to Multipoint and Multicast Semantics for additional information.

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 (for example, 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 (nonrooted 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 zero 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 **WSAConnect** function returns. For nonblocking 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 multipoint session establishment time, an application may use the *lpSQOS* and/or *lpGQOS* parameters to override any previous QOS specification made for the socket through **WSAIoctl** with either the SIO_SET_QOS or SIO_SET_GROUP_QOS opcodes.

lpSQOS specifies the flow specifications 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 specification 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 zero, respectively. A NULL value for *lpSQOS* indicates no application supplied QOS.

lpGQOS specifies the flow specifications 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 *lpSQOS->ProviderSpecific* should be set to NULL and zero, 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 (that is, 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 Values

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 nonblocking 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 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. 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.

If the return error code indicates the multipoint session join attempt failed (that is, WSAECONNREFUSED, WSAENETUNREACH, WSAETIMEDOUT) the application may call WSAJoinLeaf again for the same socket.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The specified address is already in use.
WSAEINTR	The (blocking) call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	A nonblocking **WSAJoinLeaf** call is in progress on the specified socket.
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
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 cannot 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 specifications specified in *lpSQOS* and *lpGQOS* cannot be satisfied.
WSAEPROTONOSUPPORT	The *lpCallerData* augment is not supported by the service provider.
WSAETIMEDOUT	Attempt to join timed out without establishing a multipoint session.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the multipoint session join operation cannot be completed immediately. It is possible to **select** the socket while it is connecting by **select**ing it for writing.

See Also

accept, **bind**, **select**, **WSAAccept**, **WSAAsyncSelect**, **WSAEventSelect**, **WSASocket**

WSALookupServiceBegin Quick Info

The Windows Sockets **WSALookupServiceBegin** function initiates 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 WSALookupServiceBegin (
  LPWSAQUERYSET lpqsRestrictions,
  DWORD dwControlFlags,
  LPHANDLE lphLookup
);
```

Parameters

lpqsRestrictions

[in] Contains the search criteria. See below for details.

dwControlFlags

[in] Controls the depth of the search.

LUP_DEEP	Query deep as opposed to just the first level.
LUP_CONTAINERS	Return containers only
LUP_NOCONTAINERS	Do not return any containers
LUP_FLUSHCACHE	If the provider has been caching information, ignore the cache and go query the name space itself.
LUP_NEAREST	If possible, return results in the order of distance. The measure of distance is provider specific.
LUP_RES_SERVICE	This indicates whether prime response is in the remote or local part of CSADDR_INFO structure. The other part needs to be "usable" in either case.
LUP_RETURN_ALIAS	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
LUP_RETURN_TYPE	Retrieve the type
LUP_RETURN_VERSION	Retrieve the version
LUP_RETURN_COMMENT	Retrieve the comment
LUP_RETURN_ADDR	Retrieve the addresses
LUP_RETURN_BLOB	Retrieve the private data
LUP_RETURN_ALL	Retrieve all of the information

lphLookup

[out] 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 Query-Related Data Structures for additional information.

WSAQUERYSET Field Name	Query Interpretation
dwSize	Must be set to sizeof(WSAQUERYSET). This is a versioning mechanism.
dwOutputflags	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 (that is, version must match exactly, or version must be not less than the value supplied).
lpszComment	Ignored for queries.
*dwNameSpace*1	Identifier of a single name space in which to constrain the search, or NS_ALL to include all name spaces.
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.

1 See the Important note below

Important 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 needs 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, and NIS.

Return Values

The return value is zero 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

WSANO_DATA	The name was found in the database but no data matching the given restrictions was located.
WSANOTINITIALIZED	The Windows Sockets 2 DLL has not been initialized. The application must first call **WSAStartup** before calling any Windows Sockets functions.
WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space.

See Also

WSALookupServiceEnd and **WSALookupServiceNext**

WSALookupServiceEnd Quick Info

The Windows Sockets **WSALookupServiceEnd** function 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, the end call will have the same effect as a cancel and will cause the **WSALookupServiceNext** call to return immediately.

```
INT WSALookupServiceEnd (
  HANDLE hLookup
);
```

Parameters

hLookup

[in] Handle previously obtained by calling **WSALookupServiceBegin**.

Return Values

The return value is zero 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

WSA_INVALID_HANDLE	The Handle is not valid
WSANOTINITIALIZED	The Windows Sockets 2 DLL has not been initialized. The application must first call **WSAStartup** before calling any Windows Socketsfunctions.

See Also

WSALookupServiceBegin and **WSALookupServiceNext**

WSALookupServiceNext Quick Info

The Windows Sockets **WSALookupServiceNext** function 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 function until it returns WSA_E_NOMORE, indicating that all of WSAQUERYSET has been returned.

```
INT WSALookupServiceNext (
  HANDLE hLookup,
  DWORD dwControlFlags,
  LPDWORD lpdwBufferLength,
  LPWSAQUERYSET lpqsResults
);
```

Parameters

hLookup

[in] Handle returned from the previous call to **WSALookupServiceBegin**.

dwControlFlags

[in] Flags to control the next operation. This is currently used to indicate to the provider what to do if the result set is too big for the buffer. If on the previous call to **WSALookupServiceNext** the result set was too large for the buffer, the application can choose to do one of two things on this call. First, it can choose to pass a bigger buffer and try again. Second, if it cannot or is unwilling to allocate a larger buffer, it can pass LUP_FLUSHPREVIOUS to tell the provider to throw away the last result set, which was too large, and move on to the next set for this call.

lpdwBufferLength

[in/out] On input, the number of bytes contained in the buffer pointed to by *lpresResults*. On output, if the function fails and the error is WSAEFAULT, then it contains the minimum number of bytes to pass for the *lpqsResults* to retrieve the record.

lpqsResults

[out] A pointer to a block of memory, which will contain one result set in a **WSAQUERYSET** structure on return.

Query Results

The following table describes how the query results are represented in the WSAQUERYSET structure.

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	Undefined for results.
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 Values

The return value is zero 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

WSA_E_NO_MORE	There is no more data available.
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.
WSAEFAULT	The *lpqsResults* buffer was too small to contain a **WSAQUERYSET** set.
WSA_INVALID_HANDLE	The specified Lookup handle is invalid.
WSANOTINITIALIZED	The Windows Sockets 2 DLL has not been initialized. The application must first call **WSAStartup** before calling any Windows Sockets functions.
WSANO_DATA	The name was ound 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.

See Also

WSALookupServiceBegin and **WSALookupServiceEnd**

WSANtohl Quick Info

Quick Info

Quick Info

The Windows Sockets **WSANtohl** function converts a **u_long** from network byte order to host byte order.

```
int WSANtohl (
  SOCKET s,
  u_long netlong,
  u_long FAR * lphostlong
);
```

Parameters

s

[in] A descriptor identifying a socket.

netlong

[in] A 32-bit number in network byte order.

lphostlong

[out] 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 Values

If no error occurs, **WSANtohl** returns zero. 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 function.
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

htonl, **htons**, **ntohl**, **ntohs**, **WSAHtonl**, **WSAHtons**, **WSANtohs**

WSANtohs Quick Info

Quick Info

Quick Info

The Windows Sockets **WSANtohs** function converts a **u_short** from network byte order to host byte order.

```
int WSANtohs (
  SOCKET s,
  u_short netshort,
  u_short FAR * lphostshort
);
```

Parameters

s
[in] A descriptor identifying a socket.

netshort
[in] A 16-bit number in network byte order.

lphostshort
[out] 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 Values

If no error occurs, **WSANtohs** returns zero. 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 function.
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**, **ntohl**, **ntohs**, **WSAHtonl**, **WSANtohl**, **WSAHtons**

WSARecv Quick Info

Quick Info

Quick Info

The Windows Sockets **WSARecv** function receives data from a socket.

```
int WSARecv (
  SOCKET s,
  LPWSABUF lpBuffers,
  DWORD dwBufferCount,
  LPDWORD lpNumberOfBytesRecvd,
  LPDWORD lpFlags,
  LPWSAOVERLAPPED lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

Parameters

s
[in] A descriptor identifying a connected socket.

lpBuffers
[in/out] A pointer to an array of **WSABUF** structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer.

dwBufferCount
[in] The number of **WSABUF** structures in the *lpBuffers* array.

lpNumberOfBytesRecvd
[out] A pointer to the number of bytes received by this call if the receive operation completes immediately.

lpFlags
[in/out] A pointer to flags.

lpOverlapped
[in] A pointer to a WSAOVERLAPPED structure (ignored for nonoverlapped sockets).

lpCompletionRoutine
[in] A pointer to the completion routine called when the receive operation has been completed (ignored for nonoverlapped sockets).

Remarks

This function provides functionality over and above the standard **recv** function in three important areas:

1. It can be used in conjunction with overlapped sockets to perform overlapped receive operations.
2. It allows multiple receive buffers to be specified making it applicable to the scatter/gather type of I/O.
3. 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.

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 through the completion routine or **WSAGetOverlappedResult**.

If both *lpOverlapped* and *lpCompletionRoutine* are NULL, the socket in this function will be treated as a nonoverlapped socket.

For nonoverlapped 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.

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 (for example, 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 (for example, 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 operations 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, (that is, 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 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 return error code of WSAEDISCON is used to indicate graceful closure. In any case a return 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 nonoverlapped sockets.
MSG_OOB	Process out-of-band data. (See section 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 through the *cbTransferred* parameter in the completion routine (if specified), or through 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.

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.

The 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 Values

If no error occurs and the receive operation has completed immediately, **WSARecv** returns zero. 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 function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTCONN	The socket is not connected.
WSAEINTR	The (blocking) call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAENETRESET	The connection has been broken due to the remote host resetting.
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 shut down; 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. Nonoverlapped sockets: The socket is marked as nonblocking 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 with bind, or the socket is not created with the overlapped flag.
WSAECONNABORTED	The virtual circuit was terminated due to a time-out 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**

WSARecvDisconnect Quick Info

Quick Info

Quick Info

The Windows Sockets **WSARecvDisconnect** function terminates reception on a socket, and retrieves the disconnect data if the socket is connection oriented.

```
int WSARecvDisconnect (
  SOCKET s,
  LPWSABUF lpInboundDisconnectData
);
```

Parameters

s

[in] A descriptor identifying a socket.

lpInboundDisconnectData

[out] 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.

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, the TCP window is not changed and incoming data will be accepted (but not acknowledged) until the window is exhausted. 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 zero 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 Windows Sockets provider is not required to support the use of **connect/WSAConnect** on such a socket.

Return Values

If no error occurs, **WSARecvDisconnect** returns zero. 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 function.
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 Windows Sockets 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**

WSARecvEx Quick Info

Notice This function is a Microsoft-specific extension to the Windows Sockets specification. For more information, see Microsoft Extensions and Windows Sockets 2.

The Windows Sockets **WSARecvEx** function is identical to the **recv** function, except the *flags* parameter is an in-out parameter. When a partial message is received while using datagram protocol, the MSG_PARTIAL bit is set in the *flags* parameter on return from the function.

```
int PASCAL FAR WSARecvEx (
  SOCKET s,
  char FAR * buf,
  int len,
  int *flags
);
```

Parameters

s
[in] A descriptor identifying a connected socket.

buf
[out] A buffer for the incoming data.

len
[in] The length of *buf*.

flags
[in/out] Specifies whether the message is fully or partiall received for datagram sockets.

Remarks

By default, for message-oriented transport protocols the Windows Sockets **recv** function receives a single message in each call to the function. This works fine for most cases, but there are two situations in which this is insufficient: when the application's data buffer is smaller than the message, and when the message is large and arrives in several pieces.

When the buffer is smaller than the data, as much of the message as will fit is copied into the user's buffer and **recv** returns with the error code WSAEMSGSIZE. A subsequent call to **recv** will get the next part of the message. Applications written for message-oriented transport protocols should be coded for this possibility if message sizing is not guaranteed by the application's data transfer protocol. If an application gets this error code, then the entire data buffer is filled with data.

It is more complicated when a very large message arrives a little at a time. For example, if an application sends a 1-megabyte message, the transport protocol must break up the message in order to send it over the physical network. It is theoretically possible for the transport protocol on the receiving side to buffer all the data in the message, but this would be quite expensive in terms of resources.

It would be better to allow a **recv** call to complete with only a partial message and some indication to the application that the data is only a partial message. However, the Windows Sockets **recv** function has only one output parameter: a pointer to the buffer for the incoming data. Therefore, Windows NT supplies the **WSARecvEx** function, which is identical to the **recv** function except that the *flags* parameter is an in-out parameter:

Return Values

On return from **WSARecvEx**, if a partial message was received, the MSG_PARTIAL bit is set in the *flags* parameter. If a complete message was received, MSG_PARTIAL is not set in *flags.* For a stream-oriented transport protocol, MSG_PARTIAL is never set on return from **WSARecvEx**; for stream transport protocols, this function behaves identically to the Windows Sockets **recv** function.

WSARecvFrom Quick Info

Quick Info

Quick Info

The Windows Sockets **WSARecvFrom** function receives a datagram and stores the source address.

```
int WSARecvFrom (
  SOCKET s,
  LPWSABUF lpBuffers,
  DWORD dwBufferCount,
  LPDWORD lpNumberOfBytesRecvd,
  LPDWORD lpFlags,
  struct sockaddr FAR * lpFrom,
  LPINT lpFromlen,
  LPWSAOVERLAPPED lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

Parameters

s
: [in] A descriptor identifying a socket

lpBuffers
: [in/out] A pointer to an array of **WSABUF** structures. Each **WSABUF** structure contains a pointer to a buffer and the length of the buffer.

dwBufferCount
: [in] The number of **WSABUF** structures in the *lpBuffers* array.

lpNumberOfBytesRecvd
: [out] A pointer to the number of bytes received by this call if the receive operation completes immediately.

lpFlags
: [in/out] A pointer to flags.

lpFrom
: [out] An optional pointer to a buffer which will hold the source address upon the completion of the overlapped operation.

lpFromlen
: [in/out] A pointer to the size of the *from* buffer, required only if *lpFrom* is specified.

lpOverlapped
: [in] A pointer to a WSAOVERLAPPED structure (ignored for nonoverlapped sockets).

lpCompletionRoutine
: [in] A pointer to the completion routine called when the receive operation has been completed (ignored for nonoverlapped sockets).

Remarks

This function provides functionality over and above the standard **recvfrom** function in three important areas:

1. It can be used in conjunction with overlapped sockets to perform overlapped receive operations.
2. It allows multiple receive buffers to be specified making it applicable to the scatter/gather type of I/O.
3. 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.*

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 through 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 (that is, stack-based) variables for these parameters.

If both *lpOverlapped* and *lpCompletionRoutine* are NULL, the socket in this function will be treated as a nonoverlapped socket.

For nonoverlapped sockets, the blocking semantics are identical to that of the standard **recvfrom** 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 (for example, 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, (that is, the protocol is unreliable and does not support MSG_PARTIAL), the excess data is lost, and **WSARecvFrom** generates the error WSAEMSGSIZE.

For connection-oriented sockets, **WSARecvFrom** 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 read 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 return error code of WSAEDISCONN is used to indicate graceful closure. In any case, a return 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 nonoverlapped sockets.
MSG_OOB	Process out-of-band data. (See section 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, **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* is not updated. When the overlapped operation completes the amount of data transferred is indicated either through the *cbTransferred* parameter in the completion routine (if specified), or through the *lpcbTransfer* parameter in **WSAGetOverlappedResult**. Flag values are obtained either through 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.

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.

The 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 Values

If no error occurs and the receive operation has completed immediately, **WSARecvFrom** returns zero. 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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The *lpFromlen* argument was invalid: the *lpFrom* buffer was too small to accommodate the peer address, or the *lpBuffers* argument is not totally contained in a valid part of the user address space.

WSAEINTR	The (blocking) call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking Windows Sockets 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**, or the socket is not created with the overlapped flag.
WSAENETRESET	The connection has been broken due to the remote host resetting..
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 send operations.
WSAESHUTDOWN	The socket has been shut down; 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. Nonoverlapped sockets: The socket is marked as nonblocking 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.
WSAECONNABORTED	The virtual circuit was terminated due to a time-out 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**

WSARemoveServiceClass Quick Info

The Windows Sockets **WSARemoveServiceClass** function permanently unregisters service class schema.

```
INT WSARemoveServiceClass(
  LPGUID lpServiceClassId
);
```

Parameters

lpServiceClassId

[in] Pointer to the GUID for the service class you want to remove.

Return Values

The return value is zero 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

WSAETYPE_NOT_FOUND	The specified class was not found.
WSAEACCESS	The calling routine does not have sufficient privileges to remove the Service.
WSANOTINITIALIZED	The Windows Sockets 2 DLL has not been initialized. The application must first call **WSAStartup** before calling any Windows Sockets functions.
WSAEINVAL	The specified GUID was not valid.

WSAResetEvent Quick Info

Quick Info

Quick Info

The Windows Sockets **WSAResetEvent** function resets the state of the specified event object to nonsignaled.

```
BOOL WSAResetEvent(
  WSAEVENT hEvent
);
```

Parameters

hEvent

[in] Identifies an open event object handle.

Remarks

The state of the event object is set to be nonsignaled.

Return Values

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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 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

WSACloseEvent, **WSACreateEvent**, **WSASetEvent**

WSASend Quick Info

Quick Info

Quick Info

The Windows Sockets **WSASend** function sends data on a connected socket.

```
int WSASend (
  SOCKET s,
  LPWSABUF lpBuffers,
  DWORD dwBufferCount,
  LPDWORD lpNumberOfBytesSent,
  DWORD dwFlags,
  LPWSAOVERLAPPED lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

Parameters

s
[in] A descriptor identifying a connected socket.

lpBuffers
[in] 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.

dwBufferCount
[in] The number of **WSABUF** structures in the *lpBuffers* array.

lpNumberOfBytesSent
[out] A pointer to the number of bytes sent by this call if the I/O operation completes immediately.

dwFlags
[in] Specifies the way in which the call is made.

lpOverlapped
[in] A pointer to a WSAOVERLAPPED structure (ignored for nonoverlapped sockets).

lpCompletionRoutine
[in] A pointer to the completion routine called when the send operation has been completed (ignored for nonoverlapped sockets).

Remarks

This function provides functionality over and above the standard **send** function in two important areas:

1. It can be used in conjunction with overlapped sockets to perform overlapped send operations.
2. 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 through 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 nonoverlapped 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 through the completion routine or **WSAGetOverlappedResult**.

For nonoverlapped 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 nonblocking 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 is completed 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 Windows Sockets service provider can choose to ignore this flag.
MSG_OOB	Send out-of-band data on a stream-style socket such as SOCK_STREAM only. (See section Out-Of-Band data for a discussion of this topic.)
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 through the *cbTransferred* parameter in the completion routine (if specified), or through 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.

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.

The 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 Values

If no error occurs and the send operation has completed immediately, **WSASend** returns zero. 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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the

	appropriate flag was not set.
WSAEINTR	The (blocking) call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *lpBuffers* argument is not totally contained in a valid part of the user address space.
WSAENETRESET	The connection has been broken due to the remote host resetting.
WSAENOBUFS	The Windows Sockets 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 shut down; 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. Nonoverlapped sockets: The socket is marked as nonblocking 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 terminated due to a time-out 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**

WSASendDisconnect Quick Info

Quick Info

Quick Info

The Windows Sockets **WSASendDisconnect** function initiates termination of the connection for the socket and sends disconnect data.

```
int WSASendDisconnect (
  SOCKET s,
  LPWSABUF lpOUT boundDisconnectData
);
```

Parameters

s

[in] A descriptor identifying a socket.

lpOutboundDisconnectData

[in] 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.

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 Windows Sockets provider is not required to support the use of **connect/WSAConnect** on such a socket.

Return Values

If no error occurs, **WSASendDisconnect** returns zero. 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 function.
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 Windows Sockets 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**

WSASendTo Quick Info

Quick Info

Quick Info

The Windows Sockets **WSASendTo** function sends data to a specific destination, using overlapped I/O where applicable.

```
int WSASendTo (
  SOCKET s,
  LPWSABUF lpBuffers,
  DWORD dwBufferCount,
  LPDWORD lpNumberOfBytesSent,
  DWORD dwFlags,
  const struct sockaddr FAR * lpTo,
  int iToLen,
  LPWSAOVERLAPPED lpOverlapped,
  LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionROUTINE
);
```

Parameters

s
: [in] A descriptor identifying a connected socket which was created using **WSASocket** with flag WSA_FLAG_OVERLAPPED.

lpBuffers
: [in] 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.

dwBufferCount
: [in] The number of **WSABUF** structures in the *lpBuffers* array.

lpNumberOfBytesSent
: [out] A pointer to the number of bytes sent by this call if the I/O operation completes immediately.

dwFlags
: [in] Specifies the way in which the call is made.

lpTo
: [in] An optional pointer to the address of the target socket.

iToLen
: [in] The size of the address in *lpTo*.

lpOverlapped
: [in] A pointer to a WSAOVERLAPPED structure (ignored for nonoverlapped sockets).

lpCompletionRoutine
: [in] A pointer to the completion routine called when the send operation has been completed (ignored for nonoverlapped sockets).

Remarks

This function provides functionality over and above the standard **sendto** function in two important areas:

1. It can be used in conjunction with overlapped sockets to perform overlapped send operations.
2. 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. 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 nonoverlapped 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 through the completion routine or **WSAGetOverlappedResult**.

For nonoverlapped 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 nonblocking 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 is completed in an overlapped manner, it is the sercvice 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 Windows Sockets service provider can 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 through the *cbTransferred* parameter in the completion routine (if specified), or through 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.

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 Values

If no error occurs and the send operation has completed immediately, **WSASendTo** returns zero. 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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEACCES	The requested address is a broadcast address, but the appropriate flag was not set.
WSAEINTR	The (blocking) call was canceled through **WSACancelBlockingCall**.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The *lpBuffers* or *lpTo* parameters are not part of the user address space, or the *lpTo* argument is too small.
WSAENETRESET	The connection has been broken due to the remote host resetting.
WSAENOBUFS	The Windows Sockets 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 shut down; 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. Nonoverlapped sockets: The socket is marked as nonblocking 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 terminated due to a time-out or other failure.

WSAECONNRESET	The virtual circuit was reset by the remote side.
WSAEADDRNOTAVAIL	The specified address is not available from the local machine.
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAEDESTADDRREQ	A destination address is required.
WSAENETUNREACH	The network cannot 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**

WSASetBlockingHook Quick Info

This function has been removed in compliance with the Windows Sockets 2 specification, revision 2.2.0.

The function is not exported directly by the WS2_32.DLL, and Windows Sockets 2 applications should not use this function. Windows Sockets 1.1 applications that call this function are still supported through the WINSOCK.DLL and WSOCK32.DLL.

Blocking hooks are generally used to keep a single-threaded GUI application responsive during calls to blocking functions. Instead of using blocking hooks, an applications should use a separate thread (separate from the main GUI thread) for network activity.

WSASetEvent Quick Info

Quick Info

Quick Info

The Windows Sockets **WSASetEvent** function sets the state of the specified event object to signaled.

```
BOOL WSASetEvent(
  WSAEVENT hEvent
);
```

Paramters

hEvent

[in] Identifies an open event object handle.

Remarks

The state of the event object is set to be signaled.

Return Values

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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 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

WSACloseEvent, **WSACreateEvent**, **WSAResetEvent**

WSASetLastError Quick Info

The Windows Sockets **WSASetLastError** function sets the error code which can be retrieved through the **WSAGetLastError** function.

```
void WSASetLastError (
  int iError
);
```

Parameters

iError

[in] 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 Windows Sockets 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 Values

None.

Error Codes

WSANOTINITIALISED	A successful **WSAStartup** must occur before using this function.

See Also

getsockopt, **WSAGetLastError**

WSASetService Quick Info

The Windows Sockets **WSASetService** function registers or deregisters a service instance within one or more name spaces. This function can 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 WSASetService(
  LPWSAQUERYSET lpqsRegInfo,
  EWSASETSERVICEOP essOperation,
  DWORD dwControlFlags
);
```

Parameters

lpqsRegInfo

[in] Specifies service information for registration, identifies service for deregistration.

essOperation

[in] 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

[in] The meaning of *dwControlFlags* is dependent on the value of *essOperation* as follows:

essOperation	dwControlFlags	Meaning
REGISTER	SERVICE_MULTIPLE	The service being registered may be represented by multiple CSADDR_INFO structures

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
Field Name	Service Property Description
dwSize	Must be set to sizeof(WSAQUERYSET). This is a versioning mechanism.
dwOutputFlags	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 structures referenced by *lpcsaBuffer*.
lpcsaBuffer	A pointer to an array of CSADDRO_INFO structures which contain the address9(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 Values

The return value is zero 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

WSAEACCESS	The calling routine does not have sufficient privileges to install the Service.
WSANOTINITIALIZED	The Windows Sockets 2 DLL has not been initialized. The application must first call **WSAStartup** before calling any Windows Sockets functions.

WSASocket Quick Info

Quick Info

Quick Info

The Windows Sockets **WSASocket** function creates a socket which is bound to a specific transport service provider, and optionally creates and/or joins a socket group.

```
SOCKET WSASocket (
  int af,
  int type,
  int protocol,
  LPWSAPROTOCOL_INFO lpProtocolInfo,
  GROUP g,
  DWORD dwFlags
);
```

Parameters

af
[in] An address family specification.

type
[in] A type specification for the new socket.

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

lpProtocolInfo
[in] A pointer to a WSAPROTOCOL_INFO structure that defines the characteristics of the socket to be created.

g
[in] The identifier of the socket group.

dwFlags
[in] 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 socket will *not* have an overlapped attribute. If *lpProtocolInfo* is NULL, the Windows Sockets 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 structure. 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 structure (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 through the corresponding parameters to the **WSPSocket** function in the SPI.

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.

Parameter *g* is used to indicate the appropriate actions on socket groups:

1. 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

2. if *g* = SG_UNCONSTRAINED_GROUP, create an unconstrained socket group and have the new socket be the first member; or
3. if *g* = SG_CONSTRAINED_GROUP, create a constrained socket group and have the new socket be the first member; or
4. 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.
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 structure. Refer to 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 structure. Refer to 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 structure. Refer to 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 structure. Refer to Multipoint and Multicast Semantics for additional information.

Important 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 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 encourages to support raw sockets whenever it makes sense to do so.

Shared Sockets

When a special WSAPROTOCOL_INFO structure (obtained through 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**.

Return Values

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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEAFNOSUPPORT	The specified address family is not supported.
WSAEINPROGRESS	A blocking Windows Sockets 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.

See Also

accept, **bind**, **connect**, **getsockname**, **getsockopt**, **ioctlsocket**, **listen**, **recv**, **recvfrom**, **select**, **send**, **sendto**, **setsockopt**, **shutdown**

WSAStartup Quick Info

The Windows Sockets **WSAStartup** function initiates use of the Windows Sockets DLL by a process.

```
int WSAStartup (
  WORD wVersionRequested,
  LPWSADATA lpWSAData
);
```

Parameters

wVersionRequested

[in] The highest version of Windows Sockets 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

[out] A pointer to the **WSADATA** data structure that is to receive details of the Windows Sockets implementation.

Remarks

This function *must* be the first Windows Sockets function called by an application or DLL. It allows an application or DLL to specify the version of Windows Sockets required and to retrieve details of the specific Windows Sockets implementation. The application or DLL may only issue further Windows Sockets functions after a successful **WSAStartup** invocation.

In order to support future Windows Sockets implementations and applications which may have functionality differences from current version of Windows Sockets, a negotiation takes place in **WSAStartup**. The caller of **WSAStartup** and the Windows Sockets 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 Windows Sockets 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 Windows Sockets 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 Windows Sockets DLL or fail to initialize.

This negotiation allows both a Windows Sockets DLL and a Windows Sockets application to support a range of Windows Sockets versions. An application can successfully utilize a Windows Sockets 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 Windows Sockets DLL versions:

App versions	DLL Versions	*wVersion Requested*	*wVersion*	*wHigh Version*	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	---	---	WSAVERNOT SUPPORTED
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 of Windows

Sockets makes a **WSAStartup** call:

```
WORD wVersionRequested;
WSADATA wsaData;
int err;

wVersionRequested = MAKEWORD( 2, 0 );

err = WSAStartup( wVersionRequested, &wsaData );
if ( err != 0 ) {
    /* Tell the user that we couldn't find a usable */
    /* WinSock DLL.                                  */
    return;
}

/* Confirm that the WinSock DLL supports 2.0.*/
/* Note that if the DLL supports versions greater    */
/* than 2.0 in addition to 2.0, it will still return */
/* 2.0 in wVersion since that is the version we      */
/* requested.                                        */

if ( LOBYTE( wsaData.wVersion ) != 2 ||
        HIBYTE( wsaData.wVersion ) != 0 ) {
    /* Tell the user that we couldn't find a usable */
    /* 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 Windows Sockets calls as needed. When it has finished using the services of the Windows Sockets DLL, the application or DLL must call **WSACleanup** in order to allow the Windows Sockets DLL to free any resources for the application.

Details of the actual Windows Sockets 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:

Parameters

wVersion

The version of the Windows Sockets specification that the Windows Sockets DLL expects the caller to

use.

wHighVersion
: The highest version of the Windows Sockets specification that this DLL can support (also encoded as above). Normally this will be the same as *wVersion*.

sz
: A null-terminated ASCII string into which the Windows Sockets DLL copies a description of the Windows Sockets 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 Windows Sockets DLL copies relevant status or configuration information. The Windows Sockets 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 backward compatibility, but should be ignored for version 2 and later as no single value can be appropriate for all underlying service providers.

iMaxUdpDg
: This value should be ignored for version 2 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 Windows Sockets 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 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. This is because the architecture of Windows Sockets has been changed in version 2 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 Windows Sockets 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 Values

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 Windows Sockets 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 Windows Sockets support requested is not provided by this particular Windows Sockets implementation.
WSAEINPROGRESS	A blocking Windows Sockets 1.1 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**

WSAStringToAddress Quick Info

The Windows Sockets **WSAStringToAddress** function converts a human-readable string to a socket address structure (SOCKADDR) suitable to pass 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 default to zero. If the caller wants the translation to be done by a particular provider, it should supply the corresponding WSAPROTOCOL_INFO structure in the *lpProtocolInfo* parameter.

```
INT WSAStringToAddress(
  LPTSTR AddressString,
  INT AddressFamily,
  LPWSAPROTOCOL_INFO lpProtocolInfo,
  LPSOCKADDR lpAddress,
  LPINT lpAddressLength
);
```

Parameters

AddressString
[in] Points to the zero-terminated human-readable string to convert.

AddressFamily
[in] The address family to which the string belongs.

lpProtocolInfo
[in] (Optional) The WSAPROTOCOL_INFO structure for a particular provider.

Address
[in/out] A buffer which is filled with a single **SOCKADDR** structure.

lpAddressLength
[in/out] The length of the Address buffer. Returns the size of the resultant **SOCKADDR** structure.

Return Values

The return value is zero 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

WSAEFAULT	The specified Address buffer is too small. Pass in a larger buffer.
WSAEINVAL	Unable to translate the string into a **SOCKADDR**.

WSAUnhookBlockingHook Quick Info

This function has been removed in compliance with the Windows Sockets 2 specification, revision 2.2.0.

The function is not exported directly by the WS2_32.DLL, and Windows Sockets 2 applications should not use this function. Windows Sockets 1.1 applications that call this function are still supported through the WINSOCK.DLL and WSOCK32.DLL.

Blocking hooks are generally used to keep a single-threaded GUI application responsive during calls to blocking functions. Instead of using blocking hooks, an applications should use a separate thread (separate from the main GUI thread) for network activity.

WSAWaitForMultipleEvents Quick Info

Quick Info

Quick Info

The Windows Sockets **WSAWaitForMultipleEvents** function returns either when one or all of the specified event objects are in the signaled state, or when the time-out interval expires.

```
DWORD WSAWaitForMultipleEvents(
  DWORD cEvents,
  const WSAEVENT FAR *lphEvents,
  BOOL fWaitAll,
  DWORD dwTimeOUT,
  BOOL fAlertable
);
```

Parameters

cEvents

[in] 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

[in] Points to an array of event object handles.

fWaitAll

[in] 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

[in] 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

[in] 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

The **WSAWaitForMultipleEvents** function 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 signalled should use the Win32 **sleepEx** function.

Return Values

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 function.
WSAENETDOWN	The network subsystem has failed.
WSAEINPROGRESS	A blocking Windows Sockets 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

WSACloseEvent, **WSACreateEvent**

AFPROTOCOLS

```
typedef struct _AFPROTOCOLS {
    INT iAddressFamily;
    INT iProtocol;
} AFPROTOCOLS, *PAFPROTOCOLS, *LPAFPROTOCOLS;
```

CSADDR_INFO Quick Info

```
typedef struct _CSADDR_INFO {
    SOCKET_ADDRESS LocalAddr ;
    SOCKET_ADDRESS RemoteAddr ;
    INT iSocketType ;
    INT iProtocol ;
} CSADDR_INFO, *PCSADDR_INFO, FAR * LPCSADDR_INFO ;
```

flowspec

```
typedef struct _flowspec
{
    int32        TokenRate;            /* In Bytes/sec    */
    int32        TokenBucketSize;      /* In Bytes        */
    int32        PeakBandwidth;        /* In Bytes/sec    */
    int32        Latency;              /* In microseconds */
    int32        DelayVariation;       /* In microseconds */
    GUARANTEE    LevelOfGuarantee;     /* Guaranteed, Predictive, */
                                       /* Best Effort, etc.  */
    int32        CostOfCall;           /* Reserved for future use, */
                                       /* must be set to 0 now     */
    int32        NetworkAvailability;  /* read-only:         */
                                       /*   1 if accessible, */
                                       /*   0 if not         */
} FLOWSPEC, FAR * LPFLOWSPEC;
```

Members

TokenRate

A *token bucket model* is used to specify the rate at which permission to send traffic (or credits) accrues. In the model, the token bucket has a maximum volume, TokenBucketSize, and continuously fills at a certain rate TokenRate. If the bucket contains sufficient credit, the application can send data and reduce the available credit by that amount. If sufficient credits are not available, the application must wait or discard the extra traffic.

A value of -1 in the members TokenRate and TokenBucketSize indicates that no rate-limiting is in force. The TokenRate is expressed in bytes per second.

If an application has been sending at a low rate for a period of time, it can 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. In constant rate applications, the TokenRate is equal to the PeakBandwidth.

TokenBucketSize

The *TokenBucketSize* is expressed in bytes.

The TokenBucketSize is the largest typical frame size in video applications. In constant rate applications, the TokenBucketSize is chosen to accommodate small variations.

PeakBandwidth

This member, 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 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.

LevelOfGuarantee

This is the level of service being negotiated for. The GUARANTEE type is enumerated below. Four levels of service are defined: Guaranteed, Guaranteed Delay, Predictive, Controlled Load and Best Effort.

The reason for defining both predictive and guaranteed service is that predictive services may achieve substantially better performance given the same level of network resource usage, while guaranteed service provides the mathematical level of certainty needed by selected applications. Specific providers may implement none, one, or both of these services.

Best effort service is just a hint to the service provider and should be always supported.

CostOfCall

This is just a place holder for now and should always be set to 0 until we can come up with a meaningful cost metric.

NetworkAvailability

Network Availability - This is a read-only field for the transport provider to use in indicating to the application whether or not the underlying media is currently accessible or temporarily unavailable. The typical example for a temporarily inaccessible network would be a wireless interface that has lost contact with the base station (due, for example to terrain interference). Any change in this value should result in an FD_QOS indication to applications that have registered interest in same.

FD_SET

```
typedef struct fd_set {
    u_int fd_count;                 /* how many are SET? */
    SOCKET fd_array[FD_SETSIZE];   /* an array of SOCKETs */
} fd_set;

typedef struct fd_set FD_SET;
```

Element	Usage
fd_count	The number of sockets that are set.
fd_array	An array of the sockets in the set.

GUARANTEE

```
typedef enum
{
    BestEffortService,
    ControlledLoadService,
    PredictiveService,
    GuaranteedService
} GUARANTEE;
```

Types

GuaranteedService

A service provider supporting guaranteed service 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 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.

PredictiveService

A service provider supporting predictive service 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. The delay limit is not guaranteed (occasional packets may take longer than specified), but is generally highly reliable. This service is designed for applications that can accommodate or adapt to some variation in service quality, such as video service.

ControlledLoadService

With this service, 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 can 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).
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.

Note This definition comes from the Internet Engineering Task Force (IETF).

BestEffortService

A service provider supporting best effort service, 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.

hostent

This structure is allocated by Windows Sockets. An application should 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 that it needs before issuing any other Windows Sockets API calls.

```
struct hostent {
    char FAR *       h_name;
    char FAR * FAR * h_aliases;
    short            h_addrtype;
    short            h_length;
    char FAR * FAR * h_addr_list;
};
```

Members

h_name
Official name of the host (PC).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.

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.

QualityOfService

The Windows Sockets 2 QOS structure is defined in WINSOCK2.H and is reproduced here.

```
typedef struct _QualityOfService
{
    FLOWSPEC    SendingFlowspec;       /* The flowspec for data */
                                       /*   sending             */
    FLOWSPEC    ReceivingFlowspec;     /* The flowspec for data */
                                       /*   receiving           */
    WSABUF      ProviderSpecific;      /* Additional provider-  */
                                       /*   specific parameters */
} QOS, FAR * LPQOS;
```

sockaddr

The **sockaddr** structure varies depending on the the protocol selected. The structure below is used with TCP/IP. Other protocols use similar structures.

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

SOCKET_ADDRESS

```
typedef struct _SOCKET_ADDRESS {
    LPSOCKADDR lpSockaddr ;
    INT iSockaddrLength ;
} SOCKET_ADDRESS, *PSOCKET_ADDRESS, FAR * LPSOCKET_ADDRESS ;
```

WSABUF

```
typedef struct _WSABUF {
    u_long      len;
    char FAR *  buf;
} WSABUF, FAR * LPWSABUF;
```

Members

len

The length of the buffer.

buf

A pointer to the buffer.

WSAEcomparator

```
typedef enum _WSAEcomparator
{
    COMP_EQUAL = 0,
    COMP_NOTLESS
} WSAECOMPARATOR, *PWSAECOMPARATOR, *LPWSAECOMPARATOR;
```

WSAData

```
typedef struct WSAData {
        WORD                    wVersion;
        WORD                    wHighVersion;
        char                    szDescription[WSADESCRIPTION_LEN+1];
        char                    szSystemStatus[WSASYS_STATUS_LEN+1];
        unsigned short          iMaxSockets;
        unsigned short          iMaxUdpDg;
        char FAR *              lpVendorInfo;
} WSADATA, FAR * LPWSADATA;
```

WSANAMESPACE_INFO

The **WSANAMESPACE_INFO** structure contains all of the registration information for a name space provider.

```
typedef struct _WSANAMESPACE_INFOW {
    GUID               NSProviderId;
    DWORD              dwNameSpace;
    BOOL               fActive;
    DWORD              dwVersion;
    LPWSTR             lpszIdentifier;
} WSANAMESPACE_INFOW, *PWSANAMESPACE_INFOW, *LPWSANAMESPACE_INFOW;
```

Members

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.

WSAOVERLAPPED

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;
    DWORD        InternalHigh;
    DWORD        Offset;
    DWORD        OffsetHigh;
    WSAEVENT     hEvent;
} WSAOVERLAPPED, LPWSAOVERLAPPED;
```

Members

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
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.

OffsetT
This field is reserved for service providers to use.

OffsetHigh
This field is reserved for service providers to use.

Event
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.

WSAQuerySet

```
typedef struct _WSAQuerySetW
{
    DWORD           dwSize;
    LPWSTR          lpszServiceInstanceName;
    LPGUID          lpServiceClassId;
    LPWSAVERSION    lpVersion;
    LPWSTR          lpszComment;
    DWORD           dwNameSpace;
    LPGUID          lpNSProviderId;
    LPWSTR          lpszContext;
    DWORD           dwNumberOfProtocols;
    LPAFPROTOCOLS   lpafpProtocols;
    LPWSTR          lpszQueryString;
    DWORD           dwNumberOfCsAddrs;
    LPCSADDR_INFO   lpcsaBuffer;
    DWORD           dwOutputFlags;
    LPBLOB          lpBlob;
} WSAQUERYSETW, *PWSAQUERYSETW, *LPWSAQUERYSETW;
```

WSAPROTOCOL_INFO

```
typedef struct _WSAPROTOCOL_INFOW {
    DWORD dwServiceFlags1;
    DWORD dwServiceFlags2;
    DWORD dwServiceFlags3;
    DWORD dwServiceFlags4;
    DWORD dwProviderFlags;
    GUID ProviderId;
    DWORD dwCatalogEntryId;
    WSAPROTOCOLCHAIN ProtocolChain;
    int iVersion;
    int iAddressFamily;
    int iMaxSockAddr;
    int iMinSockAddr;
    int iSocketType;
    int iProtocol;
    int iProtocolMaxOffset;
    int iNetworkByteOrder;
    int iSecurityScheme;
    DWORD dwMessageSize;
    DWORD dwProviderReserved;
    WCHAR  szProtocol[WSAPROTOCOL_LEN+1];
} WSAPROTOCOL_INFOW, FAR * LPWSAPROTOCOL_INFOW;
```

Members

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. 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.

dwServiceFlags2
Reserved for additional protocol attribute definitions.

dwServiceFlags3
Reserved for additional protocol attribute definitions.

dwServiceFlags4
Reserved for additional protocol attribute definitions.

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 WS2_32.DLL that this protocol should not be returned in the result buffer generated by **WSAEnumProtocols**. Obviously, a Windows Sockets 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.

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 *dwProviderId* value to distinguish between providers that might otherwise be indistinguishable.

dwCatalogEntryId

A unique identifier assigned by the WS2_32.DLL for each WSAPROTOCOL_INFO structure.

WSAPROTOCOLCHAIN ProtocolChain;

If the length of the chain is 0, this WSAPROTOCOL_INFO entry represents a layered protocol which has Windows Sockets 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 of the WSAPROTOCOL_INFO structure. 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 zero element in the ProtocolChain.ChainEntries array) and ending with the base protocol. Refer to the Windows Sockets 2 Service Provider Interface specification for more information on protocol chains.

iVersion

Protocol version identifier.

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.

iMaxSockAddr

The maximum address length.

iMinSockAddr

The minimum address length.

iSocketType

The value to pass as the socket type parameter to the **socket** function in order to open a socket for this protocol.

iProtocol

The value to pass as the protocol parameter to the **socket** function in order to open a socket for this protocol.

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.

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.

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.

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. 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 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.

dwProviderReserved

Reserved for use by service providers.

szProtocol

An array of characters that contains a human-readable name identifying the protocol, for example "SPX2". The maximum number of characters allowed is WSAPROTOCOL_LEN, which is defined to be 255.

WSAPROTOCOLCHAIN

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

```
typedef struct _WSAPROTOCOLCHAIN {
    int ChainLen;           /* the length of the chain,  */
                            /* length = 0 means layered protocol, */
                            /* length = 1 means base protocol,
                            /* length > 1 means protocol chain */
    DWORD ChainEntries[MAX_PROTOCOL_CHAIN];
                            /* a list of dwCatalogEntryIds */
} WSAPROTOCOLCHAIN, FAR * LPWSAPROTOCOLCHAIN;
```

WSAServiceClassInfo

```
typedef struct _WSAServiceClassInfoW
{
    LPGUID              lpServiceClassId;
    LPWSTR              lpszServiceClassName;
    DWORD               dwCount;
    LPWSANSCLASSINFOW   lpClassInfos;
}WSASERVICECLASSINFOW, *PWSASERVICECLASSINFOW, *LPWSASERVICECLASSINFOW;
```