

# Legal Information

## Windows Sockets 2 Application Programming Interface

This is a preliminary document and may be changed substantially prior to final commercial release. This document is provided for informational purposes only and Microsoft makes no warranties, either express or implied, in this document. Information in this document is subject to change without notice. The entire risk of the use or the results of the use of this document remains with the user. The names of companies, products, people, characters, and/or data mentioned herein are fictitious and are in no way intended to represent any real individual, company, product, or event, unless otherwise noted. Complying with all applicable copyright laws is the responsibility of the user. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Microsoft Corporation. If, however, your only means of access is electronic, permission to print one copy is hereby granted.

Portions of this document specify and accompany software that is still in development. Some of the information in this documentation may be inaccurate or may not be an accurate representation of the functionality of final documentation or software. Microsoft assumes no responsibility for any damages that might occur directly or indirectly from these inaccuracies.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

©1996-1998 Microsoft Corporation. All rights reserved.

Microsoft, MS, Win32, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Intel is a registered trademark of Intel Corporation.

All other product and company names mentioned herein are the trademarks of their respective owners.

## 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 Platform 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	Windows Sockets 2 establishes conventions applications use to negotiate required service levels for parameters such as bandwidth and latency. Other QOS-related enhancements include 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 Platform 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

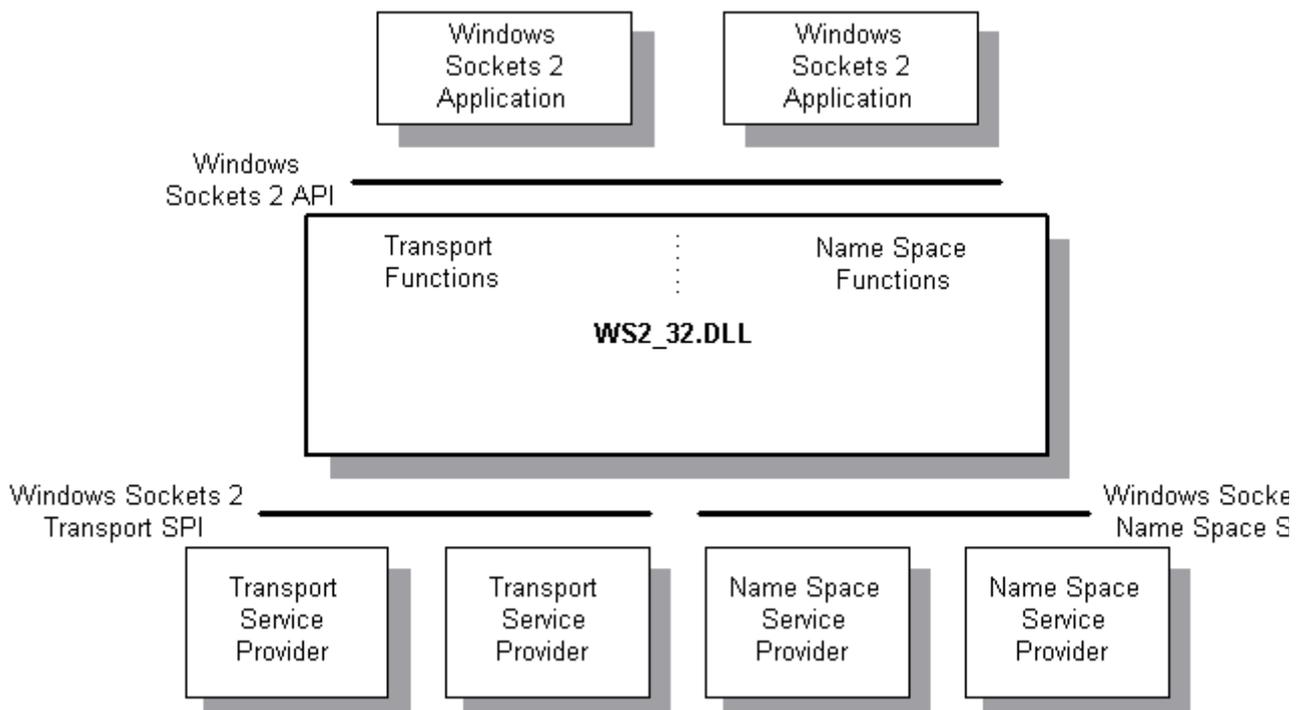
A number of Windows Sockets 2 features required a substantial change in the Windows Sockets architecture. The resulting architecture is considerably different from previous versions, but the benefits are numerous. Foremost among these is [Simultaneous Access to Multiple Transport Protocols](#), explained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



## 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

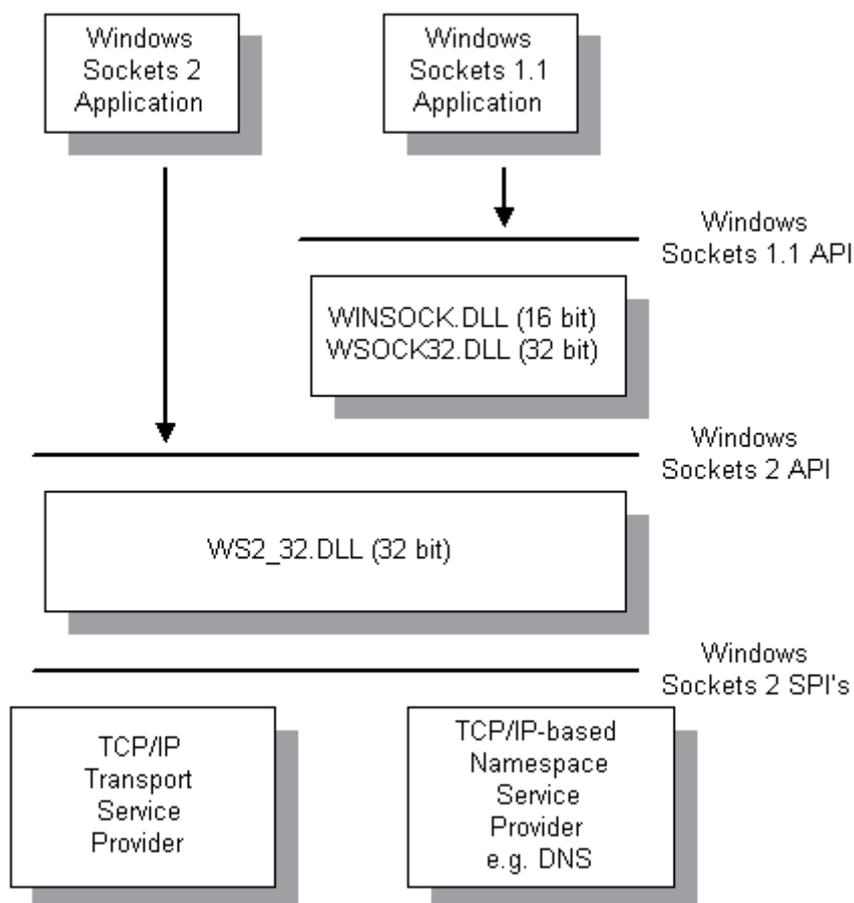
Source code compatibility in Windows Sockets 2 means, with few exceptions, that all the Windows Sockets 1.1 functions are preserved in Windows Sockets 2. Windows Sockets 1.1 applications that make use of blocking hooks will need to be modified since blocking hooks are no longer supported in Windows Sockets 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.



## 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 [Welcome To Windows Sockets 2 SPI](#) 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, SPORDER.EXE, that allows the user to view and modify the order in which service providers are enumerated. By using this SPORDER.EXE, 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.

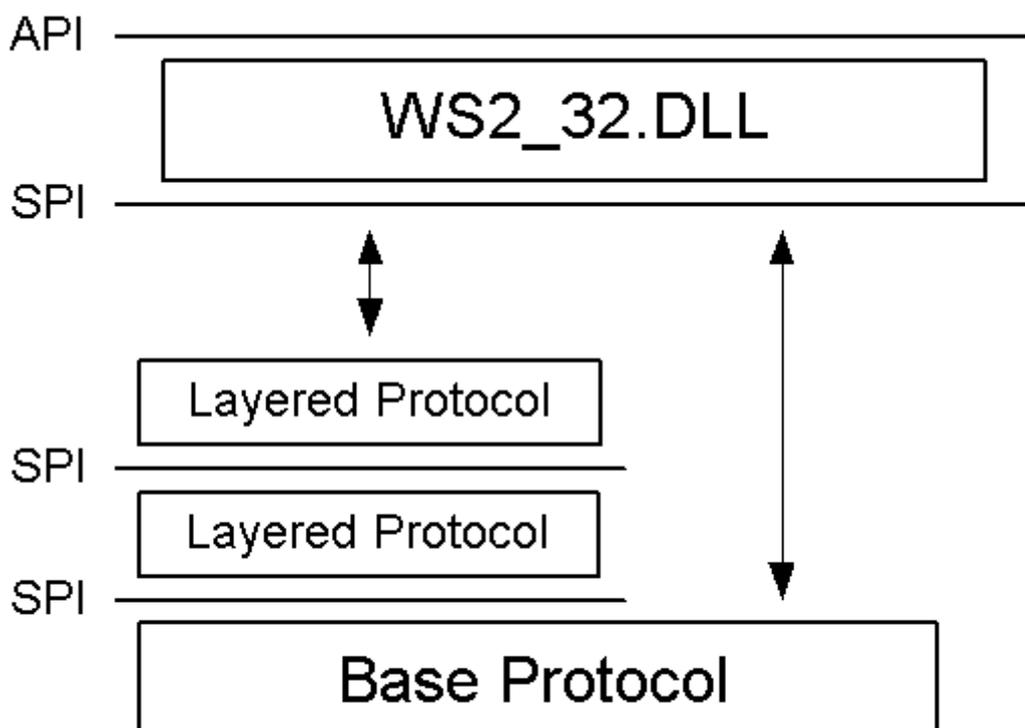
The SPORDER.EXE applet exports functions from SPORDER.DLL to reorder the service providers. As a result, installation applications can use the interface of SPORDER.DLL to programmatically reorder service providers to suit their needs.

### 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.



---

## 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

Windows Sockets 2 application developers need to isolate bugs in 1) the application, 2) the WS2\_32.DLL or one of the compatibility "shim" DLLs, 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 created through the [WSASocket](#) function with the WSA\_FLAG\_OVERLAPPED flag set or sockets created through the [socket](#) function.

**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 application 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.

Canceling individual overlapped operations pending on a given socket is impossible. However, the [closesocket](#) function can be called to close the socket and eventually discontinue all pending operations.

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

## 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](#), and [WSAWaitForMultipleEvents](#) 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. An application that uses the completion routine option for a particular overlapped I/O request may not use the "wait" option of [WSAGetOverlappedResult](#) for that same overlapped I/O request.

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`:

<i>lpOverlapped</i>	<i>hEvent</i>	<i>lpCompletionRoutine</i>	<b>Completion Indication</b>
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.

## Flow Specification 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.

#### 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. The documentation for the [WSAGetQOSByName](#) function will be periodically updated with a list of flow specifications and general descriptions as they become well-known. The [WSAGetQOS ByName](#) function can also be used to enumerate the set of known QOS template names.

## 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 =           0xFFFFFFFF(not specified)
TokenBucketSize =    0xFFFFFFFF(not specified)
PeakBandwidth =      0xFFFFFFFF(not specified)
Latency =             0xFFFFFFFF(not specified)
DelayVariation =     0xFFFFFFFF(not specified)
ServiceType =        SERVICETYPE
MaxSduSize =...      0xFFFFFFFF
MinimumPolicedSize = 0xFFFFFFFF
```

## Socket Groups

**Important** Reserved for future use with socket groups:

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

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 for a target process ID. It uses some interprocess communications (IPC) mechanism to pass the contents of this structure to a target process. The target process then uses the [WSAPROTOCOL\\_INFO](#) structure in a call to [WSPSocket](#). The socket descriptor returned by this function will be an additional socket descriptor to an underlying socket which thus becomes shared. 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**

(1) Invoke [WSASendDisconnect](#) to conclude session and supply transaction total

(6) Receive FD\_CLOSE indication

(7) Invoke [WSARecvDisconnect](#) to receive and store cumulative grand total of transactions

(8) Invoke [closesocket](#)

**Server Side**

(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

(5a) 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 parameter the socket handle that has a [WSAPROTOCOL\\_INFO](#) structure associated with it. The *NetworkByteOrder* member of the [WSAPROTOCOL\\_INFO](#) structure specifies the desired network byte order (currently either "big-endian" or "little-endian").

## 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 Platform SDK).

Value	Type	Meaning	Default	Note
SO_GROUP_ID	GROUP	Reserved for future use with socket groups: The identifier of the group to which this socket belongs.	NULL	get only

SO_GROUP_PRIORITY	int	Reserved for future use with socket groups: The relative priority for sockets that are part of a socket group.	0	
SO_MAX_MSG_SIZE	int	Maximum outbound (send) size of a message for message-oriented socket types. There is no provision for finding out the maximum inbound message size. Has no meaning for stream-oriented sockets.	Implementation dependent	get only
SO_PROTOCOL_INFO	struct <a href="#">WSAPROTOCOL_INFO</a>	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](#). The other new protocol-specific ioctl opcodes can be found in the Protocol-specific Annex (a separate document included with the Platform 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 <a href="#">sockaddr</a>	<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 <b>sendto/WSASendTo</b> .
SIO_GET_QOS	<not used>	QOS	Retrieve current flow specification(s) for the socket.
SIO_GET_GROUP_QOS	<not used>	QOS	Reserved for future use with socket groups: Retrieve current group flow specification(s) for the group this socket belongs to.
SIO_MULTIPPOINT_LOOPBACK	BOOL	<not used>	Control whether data sent in a multipoint session will also be received by the same socket on the local host.
SIO_MULTICAST_SCOPE	int	<not used>	Specify the scope over which multicast transmissions will occur.
SIO_SET_QOS	QOS	<not used>	Establish new flow specification(s) for the socket.
SIO_SET_GROUP_QOS	QOS	<not used>	Reserved for future use with socket groups: 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 <i>s</i> that is valid in the context of a companion interface.
SIO_ROUTING_INTERFACE_QUERY	SOCKADDR	SOCKADDR	Obtain the address of local interface which should be used to send to the specified address
SIO_ROUTING_INTERFACE_CHANGE	SOCKADDR	<not used>	Request notification of changes in information reported via SIO_ROUTING_INTERFACE_QUERY for the specified address
SIO_ADDRESS_LIST_QUERY	<not used>	SOCKET _ADDRESS _LIST	Obtain the list of addresses to which application can bind.
SIO_ADDRESS_LIST_CHANGE	<not used>	<not used>	Request notification of changes in information reported via SIO_ADDRESS_LIST_QUERY

## Summary of New Functions

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

### Data Transport Functions

**Function**[WSAAccept1](#)[WSACloseEvent](#)[WSAConnect1](#)[WSACreateEvent](#)[WSADuplicateSocket](#)[WSAEnumNetworkEvents](#)[WSAEnumProtocols](#)[WSAEventSelect](#)[WSAGetOverlappedResult](#)[WSAGetQOSByName](#)[WSAHtonl](#)[WSAHtons](#)[WSAIoctl1](#)[WSAJoinLeaf1](#)[WSANtohl](#)[WSANtohs](#)[WSAProviderConfigChange](#)[WSARecv1](#)[WSARecvDisconnect](#)[WSARecvFrom1](#)[WSAResetEvent](#)[WSASend1](#)[WSASendDisconnect](#)[WSASendTo1](#)**Description**

An extended version of **accept** which allows for conditional acceptance and socket grouping.

Destroys an event object.

An extended version of **connect** which allows for exchange of connect data and QOS specification.

Creates an event object.

Creates a new socket descriptor for a shared socket.

Discovers occurrences of network events.

Retrieves information about each available protocol.

Associates network events with an event object.

Gets completion status of overlapped operation.

Supplies QOS parameters based on a well-known service name.

Extended version of **htonl**.

Extended version of **htons**.

Overlapped-capable version of **ioctlsocket**.

Joins a leaf node into a multipoint session.

Extended version of **ntohl**.

Extended version of **ntohs**.

Receive notifications of service providers being installed/removed.

An extended version of **recv** which accommodates scatter/gather I/O, overlapped sockets, and provides the *flags* parameter as IN OUT.

Terminates reception on a socket and retrieves the disconnect data, if the socket is connection-oriented.

An extended version of **recvfrom** which accommodates scatter/gather I/O, overlapped sockets, and provides the *flags* parameter as IN OUT.

Resets an event object.

An extended version of **send** which accommodates scatter/gather I/O and overlapped sockets.

Initiates termination of a socket connection and optionally sends disconnect data.

An extended version of **sendto** which accommodates scatter/gather I/O and overlapped sockets.

<a href="#">WSASetEvent</a>	Sets an event object.
<a href="#">WSASocket</a>	An extended version of <b>socket</b> which takes a <a href="#">WSAPROTOCOL_INFO</a> structure as input and allows overlapped sockets to be created. Also allows socket groups to be formed.
<a href="#">WSAWaitForMultipleEvents</a> <sup>1</sup>	Blocks on multiple event objects.
1	The routine can block if acting on a blocking socket.

## Name Registration and Resolution Functions

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

# 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 `SOCKET`s. 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 although this is only relevant to 16-bit applications and meaningless in a 32-bit. 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)
    {...}
```

## Service Provided 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 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 with Windows Sockets 1.1. Because this was not always possible, the psuedo-blocking facilities described below were provided.

**Note** Windows Sockets 2 only runs on pre-emptive 32-bit operating systems where deadlocks are not a problem. Programming practices for recommended for Windows Sockets 1.1 are not necessary in Windows Sockets 2.

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 16-bit 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 16-bit 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 is 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](#) function 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. For Windows Sockets 2 applications that negotiate version 2.0 or higher at the time of calling **WSAStartup**, no restriction on the nesting of operations exists. Operations can become nested

under some rare circumstances such as during a **WSAAccept** conditional-acceptance callback, or if a service provider in turn invokes a Windows Sockets 2 function.

Although this mechanism is sufficient for simple applications, it cannot support the complex message-dispatching requirements of more advanced applications (for example, those using the MDI model). For such applications, the 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

(1) Invoke **shutdown**(*s*, *SD\_SEND*) to signal end of session and that client has no more data to send.

(5') Get `FD_READ` and invoke `recv` to get any response data sent by server

(5) Receive `FD_CLOSE` indication

(6) Invoke **closesocket**

#### Server Side

(2) Receive `FD_CLOSE`, indicating graceful shutdown in progress and that all data has been received.

(3) Send any remaining response data.

(4) Invoke **shutdown**(*s*, *SD\_SEND*) to indicate server has no more data to send.

(4') 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.

### Protocol-independent 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 tables summarize the functions included in Windows Sockets 2, separated into two groups: Berkeley-style functions, and Microsoft Windows-specific Extension functions that have been ratified as part of the Windows Sockets 2 specification. These tables do not include the Windows Sockets functions known that are used with [Registration and Name Resolution](#).

### Socket Functions

The Windows Sockets specification includes all the following Berkeley-style socket routines that were part of the Windows Sockets 1.1 API:

<a href="#">accept</a> <sup>1</sup>	An incoming connection is acknowledged and associated with an immediately created socket. The original socket is returned to the listening state.
<a href="#">bind</a>	Assign a local name to an unnamed socket.
<a href="#">closesocket</a> <sup>1</sup>	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.
<a href="#">connect</a> <sup>1</sup>	Initiate a connection on the specified socket.
<a href="#">getpeername</a>	Retrieve the name of the peer connected to the specified socket.
<a href="#">getsockname</a>	Retrieve the local address to which the specified socket is bound.
<a href="#">getsockopt</a>	Retrieve options associated with the specified socket.
<a href="#">htonl</a> <sup>2</sup>	Convert a 32-bit quantity from host byte order to network byte order.
<a href="#">htons</a> <sup>2</sup>	Convert a 16-bit quantity from host byte order to network byte order.
<a href="#">inet_addr</a> <sup>2</sup>	Converts a character string representing a number in the Internet standard "." notation to an Internet address value.
<a href="#">inet_ntoa</a> <sup>2</sup>	Converts an Internet address value to an ASCII string in "." notation i.e. "a.b.c.d".
<a href="#">ioctlsocket</a>	Provide control for sockets.
<a href="#">listen</a>	Listen for incoming connections on a specified socket.
<a href="#">ntohl</a> <sup>2</sup>	Convert a 32-bit quantity from network byte order to host byte order.
<a href="#">ntohs</a> <sup>2</sup>	Convert a 16-bit quantity from network byte order to host byte order.
<a href="#">recv</a> <sup>1</sup>	Receive data from a connected or unconnected socket.

<a href="#"><u>recvfrom</u></a> 1	Receive data from either a connected or unconnected socket.
<a href="#"><u>select</u></a> 1	Perform synchronous I/O multiplexing.
<a href="#"><u>send</u></a> 1	Send data to a connected socket.
<a href="#"><u>sendto</u></a> 1	Send data to either a connected or unconnected socket.
<a href="#"><u>setsockopt</u></a>	Store options associated with the specified socket.
<a href="#"><u>shutdown</u></a>	Shut down part of a full-duplex connection.
<a href="#"><u>socket</u></a>	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](#).

<a href="#"><u>WSAAccept</u></a> 1	An extended version of accept which allows for conditional acceptance and socket grouping.
<a href="#"><u>WSAAsyncGetHostByAddr</u></a> 2 3	A set of functions which provide asynchronous versions of the standard Berkeley getXbyY functions. For example, the <b>WSAAsyncGetHostByName</b> function provides an asynchronous, message-based implementation of the standard Berkeley <b>gethostbyname</b> function.
<a href="#"><u>WSAAsyncGetHostByName</u></a> 2 3	
<a href="#"><u>WSAAsyncGetProtoByName</u></a> 2 3	
<a href="#"><u>WSAAsyncGetProtoByNumber</u></a> 2 3	
<a href="#"><u>WSAAsyncGetServByName</u></a> 2 3	
<a href="#"><u>WSAAsyncGetServByPort</u></a> 2 3	
<a href="#"><u>WSAAsyncSelect</u></a> 3	Perform asynchronous version of select
<a href="#"><u>WSACancelAsyncRequest</u></a> 2 3	Cancel an outstanding instance of a WSAAsyncGetXBY function.
<a href="#"><u>WSACleanup</u></a>	Sign off from the underlying Windows Sockets DLL.
<a href="#"><u>WSACloseEvent</u></a>	Destroys an event object.
<a href="#"><u>WSAConnect</u></a> 1	An extended version of connect which allows for exchange of connect data and QOS specification.

<a href="#"><u>WSACreateEvent</u></a>	Creates an event object.
<a href="#"><u>WSADuplicateSocket</u></a>	Allow an underlying socket to be shared by creating a virtual socket.
<a href="#"><u>WSAEnumNetworkEvents</u></a>	Discover occurrences of network events.
<a href="#"><u>WSAEnumProtocols</u></a>	Retrieve information about each available protocol.
<a href="#"><u>WSAEventSelect</u></a>	Associate network events with an event object.
<a href="#"><u>WSAGetLastError</u></a> 3	Obtain details of last Windows Sockets error
<a href="#"><u>WSAGetOverlappedResult</u></a>	Get completion status of overlapped operation.
<a href="#"><u>WSAGetQOSByName</u></a>	Supply QOS parameters based on a well-known service name.
<a href="#"><u>WSAHtonl</u></a>	Extended version of htonl
<a href="#"><u>WSAHtons</u></a>	Extended version of htons
<a href="#"><u>WSAIoctl</u></a> 1	Overlapped-capable version of ioctl
<a href="#"><u>WSAJoinLeaf</u></a> 1	Add a multipoint leaf to a multipoint session
<a href="#"><u>WSANTohl</u></a>	Extended version of ntohl
<a href="#"><u>WSANTohs</u></a>	Extended version of ntohs
<a href="#"><u>WSAProviderConfigChange</u></a>	Receive notifications of service providers being installed/removed.
<a href="#"><u>WSARecv</u></a> 1	An extended version of recv which accommodates scatter/gather I/O, overlapped sockets and provides the <i>flags</i> parameter as IN OUT
<a href="#"><u>WSARecvFrom</u></a> 1	An extended version of recvfrom which accommodates scatter/gather I/O, overlapped sockets and provides the <i>flags</i> parameter as IN OUT
<a href="#"><u>WSAResetEvent</u></a>	Resets an event object.
<a href="#"><u>WSASend</u></a> 1	An extended version of send which accommodates scatter/gather I/O and overlapped sockets
<a href="#"><u>WSASendTo</u></a> 1	An extended version of sendto which accommodates scatter/gather I/O and overlapped sockets
<a href="#"><u>WSASetEvent</u></a>	Sets an event object.
<a href="#"><u>WSASetLastError</u></a> 3	Set the error to be returned by a subsequent WSAGetLastError
<a href="#"><u>WSASocket</u></a>	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.
<a href="#"><u>WSAStartup</u></a> 3	Initialize the underlying Windows Sockets DLL.
<a href="#"><u>WSAWaitForMultipleEvents</u></a> 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.
- 3 The routine was originally a Windows Sockets 1.1 function

## 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 protocols 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 the [getservbyname](#) and [getservbyport](#) functions. Also, see [Windows Sockets 1.1 Compatible 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.

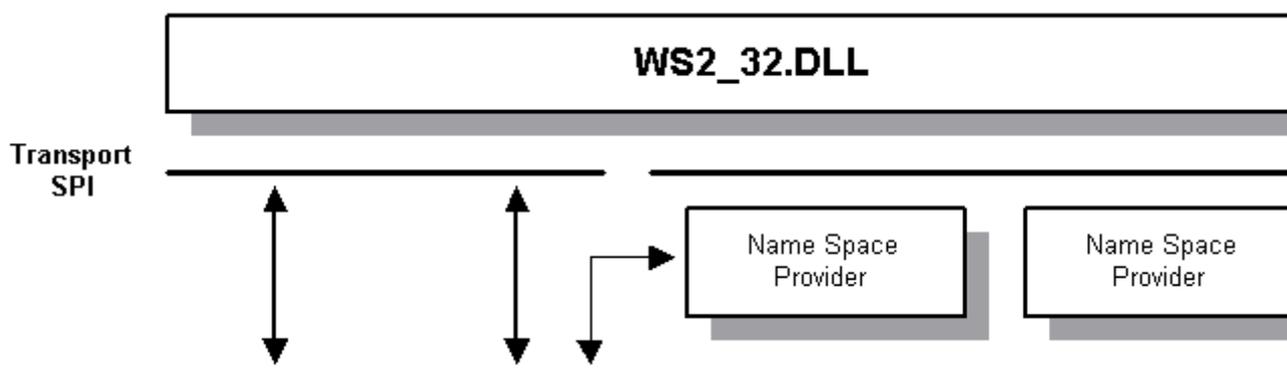
Persistent name spaces allow services to register with the name space on the fly. Unlike dynamic name spaces however, persistent name spaces retain the registration information in non-volatile storage where it remains until such time as the service requests that it be removed. Persistent name spaces 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.

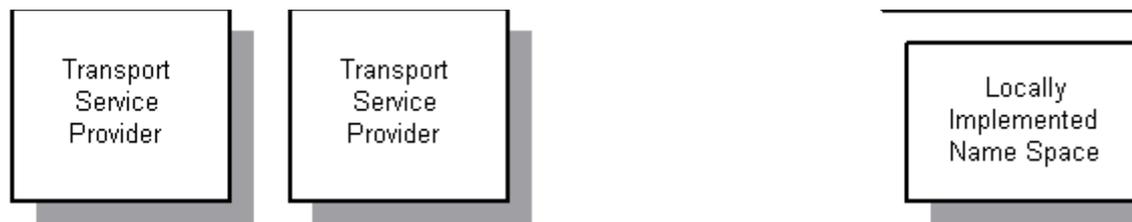
## Name Space Organization

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:





### 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 (inhost-byte order) and the corresponding class ID GUIDs. For other services, the developer chooses the class name and uses the **UUIDGEN.EXE** utility to generate a GUID for the class ID.

The notion of a service class exists to allow a set of attributes to be established that are held in common by all instances of a particular service. This set of attributes is provided at the time the service class is defined to 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 with the port expressed in host-byte order. Thus all of the familiar services such as ftp, telnet, whois, etc. are well supported.

Continuing with our service class example, instance names of the ftp service may be "alder.intel.com" or "rhino.microsoft.com" while an instance of the XYZ Corp. Employee Info Server might be named "XYZ Corp. Employee Info Server Version 3.5". In the first two cases, the combination of the service class GUID for ftp and the machine name (supplied as the service instance name) uniquely identify the desired service. In the third case, the host name where the service resides can be discovered at service query time, so the service instance name does not need to include a host name.

## 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.

In Windows Socket 2, conflicting error codes are defined for WSAENOMORE (10102) and WSA\_E\_NO\_MORE (10110). The error code WSAENOMORE will be removed in a future version and only WSA\_E\_NO\_MORE will remain. For Windows Socket 2, however, applications should check for both WSAENOMORE and WSA\_E\_NO\_MORE for the widest possible compatibility with Name Space Providers that use either one.

## Helper Functions

- [WSAGetServiceClassNameByClassId](#)
- [WSAAddressToString](#)
- [WSAStringToAddress](#)
- [WSAGetServiceClassInfo](#)

The name resolution helper functions include a function to retrieve a service class name given a service class ID, a pair of functions used to translate a transport address between a **sockaddr** struct and an ASCII string representation, a function to retrieve the service class schema information for a given service class, and a set of macros for mapping well known services to pre-allocated GUIDs.

The following macros from winsock2.h aid in mapping between well known service classes and these name spaces.

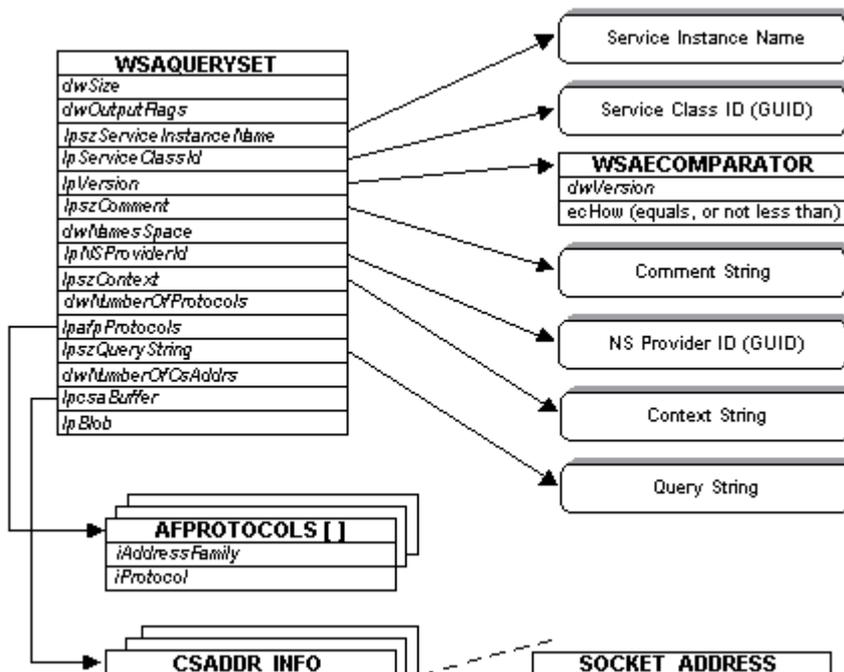
SVCID_TCP(Port)	Given a port for TCP/IP or UDP/IP or the object type in the case of Netware, return the GUID (port number in host order)
SVCID_UDP(Port)	
SVCID_NETWARE(Object Type)	Returns TRUE if the GUID is within the allowable range
IS_SVCID_TCP(GUID)	
IS_SVCID_UDP(GUID)	
IS_SVCID_NETWARE(GUID)	
SET_TCP_SVCID(GUID, port)	
SET_UDP_SVCID(GUID, port)	Initializes a GUID structure with the GUID equivalent for a TCP or UDP port number (port number must be in host order)
PORT_FROM_SVCID_TCP(GUID)	Returns the port or object type associated with the GUID (port number in host order)
PORT_FROM_SVCID_UDP(GUID)	
SAPID_FROM_SVCID_NETWARE(GUID)	

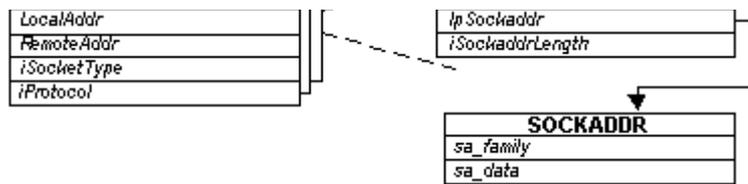
### Name Resolution Data Structures

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:





## 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 *lpzContext* 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 *lpzQueryString* 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"`.

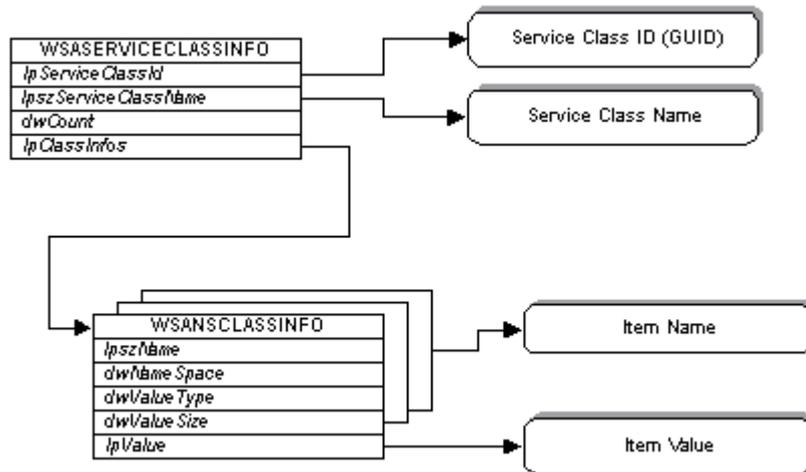
The [AFPROTOCOLS](#) structure referenced by *lpafpProtocols* is used for query purposes only, and supplies a list of protocols to constrain the query. These protocols are represented as (address family, protocol) pairs, since protocol values only have meaning within the context of an address family.

The array of [CSADDR\\_INFO](#) structure referenced by *lpcaBuffer* 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.



## 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 [WSAGetServiceClassNameByClassId](#).

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 `lpzName` 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 Compatible Name Resolution for TCP/IP

Windows Sockets 1.1 defined a number of routines that were used for name resolution with TCP/IP (IP version 4) networks. These are customarily 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 Ipv4 internet address notation to and from string and binary representations, respectively:

[inet\\_addr](#)  
[inet\\_ntoa](#)

All of these functions are specific to Ipv4 TCP/IP networks and developers of protocol-independent applications are discouraged from continuing to utilize these transport-specific functions. However, in order to retain strict backwards compatibility with 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 (these functions are not relevant to IP version 6, denoted by AF\_INET6).

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 WSALookupServiceBegin/Next/End sequence that uses one of a set of special GUIDs as the service class. These GUIDs identify the type of getXbyY operation 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 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 [WSALookupServiceBegin](#) query uses SVCID\_INET\_SERVICEBYNAME as the service class GUID. The *lpzServiceInstanceName* 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 <i>lpzServiceInstanceName</i> .
LUP_RETURN_TYPE	return canonical GUID in <i>lpServiceClassId</i> 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 <i>s_port</i> 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 <i>svcs.h</i> that corresponds to the port number indicated in the servent structure.

### The gethostbyname function

The [WSALookupServiceBegin](#) query uses SVCID\_INET\_HOSTADDRBYNAME as the service class GUID. The host name is supplied in *lpzServiceInstanceName*. 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 <i>h_name</i> field from hostent struct in <i>lpzServiceInstanceName</i> .
LUP_RETURN_ADDR	return addressing info from hostent in CSADDR_INFO structs, port information is defaulted to zero. Note that this routine does <b>not</b> resolve host names that consist of a dotted internet address.

### The gethostbyaddr function

The [WSALookupServiceBegin](#) query uses SVCID\_INET\_HOSTNAMEBYADDR as the service class GUID. The host address is supplied in *lpzServiceInstanceName* 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 <i>h_name</i> field from hostent struct in <i>lpzServiceInstanceName</i> .
LUP_RETURN_ADDR	return addressing info from hostent in CSADDR_INFO structs, port information is defaulted to zero.

### The gethostname function

The [WSALookupServiceBegin](#) query uses SVCID\_HOSTNAME as the service class GUID. If *lpzServiceInstanceName* 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 *lpzServiceInstanceName*, and specify LUP\_RETURN\_NAME so that the host name is returned in the *lpzServiceInstanceName* 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 *lpzQueryString* 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 unidirectional, 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.

	<b>rooted control plane</b>	<b>non-rooted (implicit rooted) control plane</b>
<b>rooted data plane</b>	ATM, ST-II	No known examples.
<b>non-rooted data plane</b>	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 used 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 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\_MULTIPPOINT\_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>

### **Socket Options Specific to Microsoft's Service Providers**

Microsoft's service providers support additional socket options not included in the Windows Sockets 2 API.

#### **Socket Option for NT 4.0 Only**

The following socket options are Microsoft-specific extensions for connect and disconnect data and options and are used only by non-TCP/IP transports such as DECNet, OSI TP4, etc. These are only used in Microsoft's implementation of Windows Sockets on NT 4.0.

SO\_CONNDATA

SO\_CONNOPT

SO\_DISCDATA

SO\_DISCOPT

SO\_CONNDATALEN

SO\_CONNOPTLEN

SO\_DISCDATALEN

## SO\_DISCOPTLEN

The following socket options are Microsoft-specific extensions for controlling the size of datagrams:

## SO\_MAXDG

## SO\_MAXPATHDG

## Socket Option for NT 4.0 and Win95

## SO\_SNDTIMEO

## SO\_RCVTIMEO

### Details on SO\_SNDTIMEO and SO\_RCVTIMEO

These two options set up timeouts for the [send](#), [sendto](#), [recv](#), and [recvfrom](#) functions. You can obtain the same functionality by calling [select](#) with a timeout just before the I/O call, but these options offer a significant improvement in performance by avoiding a kernel transition and the other overhead of the **select** call. For any code whose performance is very critical, applications should use these timeout options rather than **select**.

You can set these options on any type of socket in any state. The default value for these options is zero, which refers to an infinite timeout. Any other setting is the timeout, in milliseconds. It is valid to set the timeout to any value, but values less than 500 milliseconds (half a second) are interpreted to be 500 milliseconds.

To set a send timeout, use

```
int timeout = TIMEOUT_VALUE;
int err;
SOCKET s;

s = socket( ... );
err = setsockopt(
    s,
    SOL_SOCKET,
    SO_SNDTIMEO,
    (char *)&timeout,
    sizeof(timeout));
if (err != NO_ERROR) {
    /* failed for some reason... */
}
```

The `TIMEOUT_VALUE` is the needed timeout in milliseconds. To set a receive timeout, substitute `SO_RCVTIMEO` for `SO_SNDTIMEO` in the preceding example.

After setting one of these options to a nonzero value, I/O through the Windows Sockets calls fails with the error `WSAETIMEDOUT` if the request cannot be satisfied within the specified timeout. If a request times out, an application has no guarantees as to how much data was actually sent or received in the I/O call.

The following socket option is used in conjunction with the MS Extension function AcceptEx.

SO\_UPDATE\_ACCEPT\_CONTEXT

## 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*.

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. When **bind** is called with a wildcard address (involving ADDR\_ANY), a WSAEADDRINUSE error could be delayed until the specific address is "committed." This could happen with a call to other function later, including **connect**, **listen**, **WSAConnect** or **WSAJoinLeaf**.

## WSAEADDRNOTAVAIL

(10049)

*Cannot assign requested address.*

The requested address is not valid in its context. Normally results from an attempt to **bind** to an address that is not valid for the local machine. This can also result from **connect**, **sendto**, **WSAConnect**, **WSAJoinLeaf**, or **WSASendTo** when the remote address or port is not valid for a remote machine (e.g. address or port 0).

## WSAEAFNOSUPPORT

(10047)

*Address family not supported by protocol family.*

An address incompatible with the requested protocol was used. All sockets are created with an associated "address family" (i.e. AF\_INET for Internet Protocols) and a generic protocol type (i.e. SOCK\_STREAM). This error will be returned if an incorrect protocol is explicitly requested in the **socket** call, or if an address of the wrong family is used for a socket, e.g. in **sendto**.

## WSAEALREADY

(10037)

*Operation already in progress.*

An operation was attempted on a non-blocking socket that already had an operation in progress - i.e. calling **connect** a second time on a non-blocking socket that is already connecting, or canceling an asynchronous request (WSAAsyncGetXbyY) that has already been canceled or completed.

## WSAECONNABORTED

**(10053)**

*Software caused connection abort.*

An established connection was aborted by the software in your host machine, possibly due to a data transmission timeout or protocol error.

**WSAECONNREFUSED****(10061)**

*Connection refused.*

No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host - i.e. one with no server application running.

**WSAECONNRESET****(10054)**

*Connection reset by peer.*

A existing connection was forcibly closed by the remote host. This normally results if the peer application on the remote host is suddenly stopped, the host is rebooted, or the remote host used a "hard close" (see [setsockopt](#) for more information on the **SO\_LINGER** option on the remote socket.) This error may also result if a connection was broken due to "keep-alive" activity detecting a failure while one or more operations are in progress. Operations that were in progress fail with WSAENETRESET. Subsequent operations fail with WSAECONNRESET.

**WSAEDESTADDRREQ****(10039)**

*Destination address required.*

A required address was omitted from an operation on a socket. For example, this error will be returned if [sendto](#) is called with the remote address of ADDR\_ANY.

**WSAEFAULT****(10014)**

*Bad address.*

The system detected an invalid pointer address in attempting to use a pointer argument of a call. This error occurs if an application passes an invalid pointer value, or if the length of the buffer is too small. For instance, if the length of an argument which is a struct sockaddr is smaller than sizeof(struct sockaddr).

**WSAEHOSTDOWN****(10064)**

*Host is down.*

A socket operation failed because the destination host was down. A socket operation encountered a dead host. Networking activity on the local host has not been initiated. These conditions are more likely to be indicated by the error WSAETIMEDOUT.

**WSAEHOSTUNREACH****(10065)**

*No route to host.*

A socket operation was attempted to an unreachable host. See WSAENETUNREACH

**WSAEINPROGRESS****(10036)**

*Operation now in progress.*

A blocking operation is currently executing. Windows Sockets only allows a single blocking operation to be outstanding per task (or thread), and if any other function call is made (whether or not it references that or any other socket) the function fails with the WSAEINPROGRESS error.

**WSAEINTR****(10004)**

*Interrupted function call.*

A blocking operation was interrupted by a call to [WSACancelBlockingCall](#).

**WSAEINVAL****(10022)***Invalid argument.*

Some invalid argument was supplied (for example, specifying an invalid level to the [setsockopt](#) function). In some instances, it also refers to the current state of the socket - for instance, calling [accept](#) on a socket that is not **listening**.

**WSAEISCONN****(10056)***Socket is already connected.*

A connect request was made on an already connected socket. Some implementations also return this error if [sendto](#) is called on a connected SOCK\_DGRAM socket (For SOCK\_STREAM sockets, the *to* parameter in [sendto](#) is ignored), although other implementations treat this as a legal occurrence.

**WSAEMFILE****(10024)***Too many open files.*

Too many open sockets. Each implementation may have a maximum number of socket handles available, either globally, per process or per thread.

**WSAEMSGSIZE****(10040)***Message too long.*

A message sent on a datagram socket was larger than the internal message buffer or some other network limit, or the buffer used to receive a datagram into was smaller than the datagram itself.

**WSAENETDOWN****(10050)***Network is down.*

A socket operation encountered a dead network. This could indicate a serious failure of the network system (i.e. the protocol stack that the WinSock DLL runs over), the network interface, or the local network itself.

**WSAENETRESET****(10052)***Network dropped connection on reset.*

The connection has been broken due to "keep-alive" activity detecting a failure while the operation was in progress. It can 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.

**WSAENOPROTOPT****(10042)***Bad protocol option.*

An unknown, invalid or unsupported option or level was specified in a [getsockopt](#) or [setsockopt](#) call.

**WSAENOTCONN**

**(10057)**

*Socket is not connected.*

A request to send or receive data was disallowed because the socket is not connected and (when sending on a datagram socket using [sendto](#)) no address was supplied. Any other type of operation might also return this error - for example, [setsockopt](#) setting SO\_KEEPALIVE if the connection has been reset.

**WSAENOTSOCK****(10038)**

*Socket operation on non-socket.*

An operation was attempted on something that is not a socket. Either the socket handle parameter did not reference a valid socket, or for [select](#), a member of an fd\_set was not valid.

**WSAEOPNOTSUPP****(10045)**

*Operation not supported.*

The attempted operation is not supported for the type of object referenced. Usually this occurs when a socket descriptor to a socket that cannot support this operation, for example, trying to accept a connection on a datagram socket.

**WSAEPFNOSUPPORT****(10046)**

*Protocol family not supported.*

The protocol family has not been configured into the system or no implementation for it exists. Has a slightly different meaning to WSAEAFNOSUPPORT, but is interchangeable in most cases, and all Windows Sockets functions that return one of these specify WSAEAFNOSUPPORT.

**WSAEPROCLIM****(10067)**

*Too many processes.*

A Windows Sockets implementation may have a limit on the number of applications that may use it simultaneously. [WSAStartup](#) may fail with this error if the limit has been reached.

**WSAEPROTONOSUPPORT****(10043)**

*Protocol not supported.*

The requested protocol has not been configured into the system, or no implementation for it exists. For example, a [socket](#) call requests a SOCK\_DGRAM socket, but specifies a stream protocol.

**WSAEPROTOTYPE****(10041)**

*Protocol wrong type for socket.*

A protocol was specified in the [socket](#) function call that does not support the semantics of the socket type requested. For example, the ARPA Internet UDP protocol cannot be specified with a socket type of SOCK\_STREAM.

**WSAESHUTDOWN****(10058)**

*Cannot send after socket shutdown.*

A request to send or receive data was disallowed because the socket had already been shut down in that direction with a previous [shutdown](#) call. By calling [shutdown](#) a partial close of a socket is requested, which is a signal that sending or receiving or both has been discontinued.

**WSAESOCKTNOSUPPORT****(10044)**

*Socket type not supported.*

The support for the specified socket type does not exist in this address family. For example, the optional type SOCK\_RAW might be selected in a [socket](#) call, and the implementation does not support SOCK\_RAW sockets at all.

**WSAETIMEDOUT****(10060)***Connection timed out.*

A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond.

**WSATYPE\_NOT\_FOUND****(10109)***Class type not found.*

The specified class was not found.

**WSAEWOULDBLOCK****(10035)***Resource temporarily unavailable.*

This error is returned from operations on non-blocking sockets that cannot be completed immediately, for example [recv](#) when no data is queued to be read from the socket. It is a non-fatal error, and the operation should be retried later. It is normal for WSAEWOULDBLOCK to be reported as the result from calling [connect](#) on a non-blocking SOCK\_STREAM socket, since some time must elapse for the connection to be established.

**WSAHOST\_NOT\_FOUND****(11001)***Host not found.*

No such host is known. The name is not an official hostname or alias, or it cannot be found in the database(s) being queried. This error may also be returned for protocol and service queries, and means the specified name could not be found in the relevant database.

**WSA\_INVALID\_HANDLE****(OS dependent)***Specified event object handle is invalid.*

An application attempts to use an event object, but the specified handle is not valid.

**WSA\_INVALID\_PARAMETER****(OS dependent)***One or more parameters are invalid.*

An application used a 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\_INCOMPLETE****(OS dependent)***Overlapped I/O event object not in signaled state.*

The application has tried to determine the status of an overlapped operation which is not yet completed. Applications that use **WSAGetOverlappedResult** (with the *fWait* flag set to false) in a polling mode to determine when an overlapped operation has

completed will get this error code until the operation is complete.

**WSA\_IO\_PENDING**

(OS dependent)

*Overlapped operations will complete later.*

The application has initiated an overlapped operation which cannot be completed immediately. A completion indication will be given at a later time when the operation has been completed.

**WSA\_NOT\_ENOUGH\_MEMORY**

(OS dependent)

*Insufficient memory available.*

An application used a 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 the protocol/namespace catalogs.

**WSASYSNOTREADY**

(10091)

*Network subsystem is unavailable.*

This error is returned by [WSAStartup](#) if the Windows Sockets implementation cannot function at this time because the underlying system it uses to provide network services is currently unavailable. Users should check:

- that the appropriate Windows Sockets DLL file is in the current path,

- that they are not trying to use more than one 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 [WSARecv](#) and [WSARecvFrom](#) to indicate the remote party has initiated a graceful shutdown sequence.

**WSA\_OPERATION\_ABORTED****(OS dependent)***Overlapped operation aborted.*

An overlapped operation was canceled due to the closure of the socket, or the execution of the SIO\_FLUSH command in [WSAIocctl](#).

## accept

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

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

**Parameters***s*

[in] A descriptor identifying a socket that has been placed in a listening state with the [listen](#) function. The connection will actually be made with the socket that is returned by **accept**.

*addr*

[out] An optional pointer to a buffer that receives 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 established when the socket was created.

*addrLen*

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

## Remarks

The **accept** function extracts the first connection on the queue of pending connections on socket *s*. It then creates a new socket and returns a handle to the new socket. The newly created socket is the socket that will handle the actual the connection and has the same properties as socket *s*, including the asynchronous events registered with the [WSAAsyncSelect](#) or [WSAEventSelect](#) functions. The socket *s* does not have the listening socket's group ID, if any was applied.

The **accept** function can block the caller until a connection is present if no pending connections are present on the queue, and the socket is marked as blocking. If the socket is marked nonblocking and no pending connections are present on the queue, **accept** returns an error as described below. After the successful completion of **accept** returns a new socket handle, the accepted socket cannot be used to accept more connections. The original socket remains open and listens for new connection requests.

The parameter *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.

The **accept** function 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.

**Windows CE:** Windows CE does not support the WSAEINTR error value.

For IrSocket implementation, the *addr* and *addrlen* parameters should always be NULL.

## Return Values

If no error occurs, **accept** returns a value of type SOCKET that is a descriptor for the new socket. This returned value is a handle for the socket on which the actual connection will be made.

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 <a href="#">WSAStartup</a> must occur before using this FUNCTION.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The <i>addrlen</i> parameter is too small or <i>addr</i> is not a valid part of the user address space.
WSAEINTR	A blocking Windows Sockets 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .

WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEINVAL	The <a href="#">listen</a> function was not invoked prior to <b>accept</b> .
WSAEMFILE	The queue is nonempty upon entry to <b>accept</b> 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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[bind](#), [connect](#), [listen](#), [select](#), [socket](#), [WSAAsyncSelect](#), [WSAAccept](#)

# AcceptEx

**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 completed successfully and a value of `TRUE` is returned.

If the function fails, **AcceptEx** returns `FALSE`. The [WSAGetLastError](#) function can then be called to return extended error information. If **WSAGetLastError** 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 **AcceptEx** 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.

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 because the addresses are written in an internal format. For example, the size of a `SOCKADDR_IN` (the address structure for TCP/IP) is

16 bytes. Therefore, a buffer size of at least 32 bytes must be specified 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 **AcceptEx**, 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.

When the **AcceptEx** function returns, the socket *sAcceptSocket* is in the default state for a connected socket. The socket *sAcceptSocket* 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) );
```

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 a while, but no data has been received. It is recommended that you terminate 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);
}
```

## QuickInfo

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in mswsock.h.

**Import Library:** Link with mswsock.lib.

# bind

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 from the [SOCKADDR](#) structure.

*namelen*

[in] The length of the *name*.

## Remarks

The **bind** function is used on an unconnected socket before subsequent calls to the [connect](#) or [listen](#) functions. It is used to bind to either connection-oriented (stream) or connectionless (datagram) sockets. When a socket is created with a call to the [socket](#) function, it exists in a name space (address family), but it has no name assigned to it. Use **bind** to establish the local association of the socket by assigning a local name to an unnamed socket.

A name consists of three parts when using the Internet address family: the address family, a host address, and a port number that 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 1.1 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 the *sa\_family*

member of the **SOCKADDR** structure) *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, specify the manifest constant value **ADDR\_ANY** for the **sa\_data** member 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. 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. Binding to a specific port number other than port 0 is discouraged for client applications, since there is a danger of conflicting with another socket already using that port number.

**Windows CE:** For IrSocket implementation of this function:

- The `Af_irda.h` must be explicitly included.
- The **SOCKADDR\_IRDA\_wcesdk\_SOCKADDR\_IRDA** structure is used in the *addr* parameter.
- The **WSAENETDOWN** error value is not supported.
- There is no wildcard address equivalent to **INADDR\_ANY**.

**IrSockets** clients must call **bind** before using a connect function. If the service name is of the form "LSAP-SELxxx" where xxx is a decimal integer in the range 0-255, the address indicates a specific LSAP-SEL xxx rather than a service name. LSAP-SELxxx service names will cause no IAS calls. The socket will be bound directly to the LSAP-SEL specified by bypassing IAS.

## 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](#)

A process on the machine is already bound to the same fully-qualified address and the socket has not been marked to allow address re-use with **SO\_REUSEADDR**. For example, IP address and port are bound in the `af_inet` case) . (See the **SO\_REUSEADDR** socket option under [setsockopt](#).)

### [WSAEADDRNOTAVAIL](#)

The specified address is not a valid address for this machine

WSAEFAULT

The *name* or the *namelen* parameter is not a valid part of the user address space, the *namelen* parameter is too small, the *name* parameter 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.

**QuickInfo**

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

**See Also**

[connect](#), [getsockname](#), [listen](#), [setsockopt](#), [socket](#), [WSACancelBlockingCall](#)

## closesocket

The Windows Sockets **closesocket** function closes an existing socket.

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

**Parameters**

*s*

[in] A descriptor identifying a socket to close.

**Remarks**

The **closesocket** function closes a socket. Use it to release the socket descriptor *s* so 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.

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. Any event, completion routine, or completion port action specified for these overlapped operations is performed. 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 any socket resources to the system.

The semantics of **closesocket** are affected by the socket options `SO_LINGER` and `SO_DONTLINGER` as follows (`SO_DONTLINGER` is enabled by default; `SO_LINGER` is disabled):

Option	Interval	Type of close	Wait for close?
<code>SO_DONTLINGER</code>	Do not care	Graceful	No
<code>SO_LINGER</code>	Zero	Hard	No
<code>SO_LINGER</code>	Nonzero	Graceful	Yes

If `SO_LINGER` is set with a zero time-out interval (that is, the [LINGER](#) structure members **`L_onoff`** is not zero and **`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. If `SO_DONTLINGER` is set on a stream socket by setting the **`L_onoff`** member of the [LINGER](#) structure to zero, the **closesocket** call will return immediately and does not receive `WSAEWOULDBLOCK` whether the socket is blocking or nonblocking. However, any data queued for transmission will be sent, if possible, before the underlying socket is closed. This is also called a graceful disconnect. In this case, the Windows Sockets provider cannot release the socket and other resources for an arbitrary period, thus affecting applications that expect to use all available sockets. This is the default behavior (`SO_DONTLINGER` is set by default).

**Note** To assure that all data is sent and received on a connection, an application should call [shutdown](#) before calling **closesocket** (see [Graceful shutdown, linger options and socket closure](#) for more information). Also note, an `FD_CLOSE` network event 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.

**Windows CE:** Windows CE does not support the WSAEINTR error value.

For IrSocket implementation:

- The Af\_irda.h file must be explicitly included.
- The WSAENETDOWN error value is not supported.
- The standard linger options are supported.

Although IrDA does not provide a graceful close, IrSockets will defer closing until receive queues are purged. Thus, an application can send data and immediately call the **socket** function confident that the receiver will copy the data before receiving an FD\_CLOSE message.

## 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 <a href="#">WSAStartup</a> 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) Windows Socket 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .
WSAEWOULDBLOCK	The socket is marked as nonblocking and SO_LINGER is set to a nonzero time-out value.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

**See Also**

[accept](#), [ioctlsocket](#), [setsockopt](#), [socket](#), [WSAAsyncSelect](#), [WSADuplicateSocket](#)

## connect

The Windows Sockets **connect** function establishes a connection to a specified socket.

```
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 socket to connect to.

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

### Remarks

The **connect** 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](#) and [SOCKADDR](#)). When the socket call completes successfully, the socket is ready to send/receive data. If the address member of the structure specified by the *name* parameter is all zeroes, **connect** will return the error [WSAEADDRNOTAVAIL](#). Any attempt to re-connect an active connection will fail with the error code [WSAEISCONN](#).

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

For a connectionless socket (for example, type SOCK\_DGRAM), the operation performed by **connect** is merely to establish a default destination address that 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 member of the structure specified by *name* 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 use [setsockopt](#) to enable the `SO_BROADCAST` option. Otherwise, **connect** will fail with the error code [WSAEACCES](#).

When a connection between sockets is broken, the sockets should be discarded and recreated. When a problem develops on a connected socket, the application must discard and recreate the needed sockets in order to return to a stable point.

**Windows CE:** Windows CE does not support the `WSAEINTR` error value.

For IrSocket implementation:

- The `Af_irda.h` file must be explicitly included.
- If `WSAENETDOWN` is returned, an existing IrDA connection was detected at the media access level.
- If `WSAEADDRINUSE` is returned, active connections to a device with a different address exist.
- If `WSAEINPROGRESS` is returned, IAS name resolution failed because another IAS query is in progress. Retry the operation at one second intervals.
- If `WSAEISCONN` is returned, the socket is already connected or an exclusive/multiplexed mode change failed.

IrSockets implements the **connect** function with addresses of the form **sockaddr\_irda**. Typically, a client application will create a socket with the socket function, scan the immediate vicinity for IrDA devices with the `IRLMP_ENUMDEVICES` socket option, choose a device from the returned list, form an address, and call **connect**. There is no difference in blocking and non-blocking semantics.

## 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, there are three different steps you can take:

1. Use the [select](#) function to determine the completion of the connection request by checking to see if the socket is writeable.
2. If the application is using [WSAAsyncSelect](#) to indicate interest in connection events, then the application will receive an `FD_CONNECT` notification indicating that the **connect** operation is complete (successfully or not).

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

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

If the error code returned 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 <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The socket's local address is already in use and the socket was not marked to allow address reuse with <code>SO_REUSEADDR</code> . This error usually occurs when executing <b>bind</b> , but could be delayed until this function if the <b>bind</b> was to a partially wild-card address (involving <code>ADDR_ANY</code> ) and if a specific address needs to be committed at the time of this function.
WSAEINTR	The (blocking) Windows Socket 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEALREADY	A nonblocking <b>connect</b> call is in progress on the specified socket.  <b>Note</b> In order to preserve backward compatibility, this error is reported as <code>WSAEINVAL</code> to Windows Sockets 1.1 applications that link to either <code>WINSOCK.DLL</code> or <code>WSOCK32.DLL</code> .
WSAEADDRNOTAVAIL	The remote address is not a valid address (such as <code>ADDR_ANY</code> ).
WSAEAFNOSUPPORT	Addresses in the specified family cannot be used with this socket.
WSAECONNREFUSED	The attempt to connect was forcefully rejected.
WSAEFAULT	The <i>name</i> or the <i>namelen</i> parameter is not a valid part of the user address space, the <i>namelen</i> parameter is too small, or the <i>name</i> parameter contains incorrect address format for the associated address family.

WSAEINVAL	The parameter <i>s</i> 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.
WSAEACCES	Attempt to connect datagram socket to broadcast address failed because <a href="#">setsockopt</a> option SO_BROADCAST is not enabled.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[accept](#), [bind](#), [getsockname](#), [select](#), [socket](#), [WSAAsyncSelect](#), [WSAConnect](#)

# EnumProtocols

**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.

The [WSAEnumProtocols](#) function provides equivalent functionality in Windows Sockets 2.

The **EnumProtocols** function obtains information about a specified set of network protocols that are active on a local host.

```

INT EnumProtocols(
    LPINT lpProtocols,           // 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:

<b>Value</b>	<b>Protocol</b>
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** can return the following extended error code:

<b>Value</b>	<b>Meaning</b>
ERROR_INSUFFICIENT_BUFFER	The buffer pointed to by <i>lpProtocolBuffer</i> was too small to receive all of the relevant <b>PROTOCOL_INFO</b> structures. Call the function with a buffer at least as large as the value returned in <i>*lpdwBufferLength</i> .

**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;
}

```

## QuickInfo

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in nspapi.h.

**Import Library:** Link with wsock32.lib.

## See Also

[GetAddressByName](#), [PROTOCOL\\_INFO](#)

# GetAcceptExSockaddrs

**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.

## 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.

## Return Values

This function does not return a value.

## QuickInfo

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in mswsock.h.

**Import Library:** Link with mswsock.lib.

# GetAddressByName

**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.

The functions detailed in [Protocol-Independent Name Resolution](#) provide equivalent functionality in Windows Sockets 2.

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 lpProtocols,         // 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 <i>dwNameSpace</i> .
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 can access a WINS server to perform the resolution.
NS_SAP	The Netware Service Advertising Protocol. This can 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:

<b>Value</b>	<b>Meaning</b>
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 <i>lpServiceName</i> to NULL.
RES_FIND_MULTIPLE	If this flag is clear, normal name resolution occurs. 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.

*lpServiceAsyncInfo*

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** can return the following extended error value:

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

**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.

**QuickInfo**

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in nspapi.h.

**Import Library:** Link with wsock32.lib.

**See Also**

[gethostbyname](#), [CSADDR\\_INFO](#)

## gethostbyaddr

The Windows Sockets **gethostbyaddr** function retrieves the host information corresponding to a network 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

The **gethostbyaddr** function returns a pointer to the [HOSTENT](#) structure that contains the name and address corresponding to the given network address. All strings are null-terminated.

### Return Values

If no error occurs, **gethostbyaddr** returns a pointer to the **HOSTENT** structure. Otherwise, it returns a NULL pointer, and a specific error code can be retrieved by calling [WSAGetLastError](#).

### Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> 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 <i>type</i> specified is not supported by the Windows Sockets implementation.
WSAEFAULT	The <i>addr</i> parameter is not a valid part of the user address space, or the <i>len</i> parameter is too small.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[gethostbyname](#), [HOSTENT](#), [WSAAsyncGetHostByAddr](#)

# gethostbyname

The Windows Sockets **gethostbyname** function retrieves host information corresponding to a host name from a host database.

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

## Parameters

*name*

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

## Remarks

The **gethostbyname** function returns a pointer to a [HOSTENT](#) structure — a structure allocated by Windows Sockets. The **HOSTENT** structure contains the results of a successful search for the host specified in the *name* parameter.

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, so the application should copy any information it needs before issuing any other Windows Sockets function calls.

The **gethostbyname** function cannot resolve IP address strings passed to it. Such a request is treated exactly as if an unknown host name were passed. Use [inet\\_addr](#) to convert an IP address

string the string to an actual IP address, then use another function, [gethostbyaddr](#), to obtain the contents of the [HOSTENT](#) structure.

The **gethostbyname** function resolves 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 <a href="#">WSAStartup</a> 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 <i>name</i> parameter is not a valid part of the user address space.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[gethostbyaddr](#), [WSAAsyncGetHostByName](#)

## gethostname

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 receives the local host name.

*namelen*

[in] The length of the buffer.

## Remarks

The **gethostname** function 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 <i>name</i> parameter is not a valid part of the user address space, or the buffer size specified by <i>namelen</i> parameter is too small to hold the complete host name.
WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> 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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[gethostbyname](#), [WSAAsyncGetHostByName](#)

# GetNameByType

**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.

The functions detailed in [Protocol-Independent Name Resolution](#) provide equivalent functionality in Windows Sockets 2.

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](#).

## QuickInfo

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in nspapi.h.

**Import Library:** Link with wsock32.lib.

## See Also

[GetTypeByName](#)

# getpeername

The Windows Sockets **getpeername** function retrieves the name 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 that receives the name of the peer.
- namelen*  
[in/out] A pointer to the size of the *name* structure.

## Remarks

The **getpeername** function retrieves the name of the peer connected to the socket *s* and stores it in the a [SOCKADDR](#) structure identified by *name*. The **getpeername** function 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 call, the *namelen* argument contains the size of the *name* buffer, in bytes. On return, the *namelen* parameter contains the actual size in bytes of the name returned.

## 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 <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The <i>name</i> or the <i>namelen</i> parameter is not a valid part of the user address space, or the <i>namelen</i> parameter 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.

## QuickInfo

**Windows NT:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with wsock32.lib.

## See Also

[bind](#), [getsockname](#), [socket](#)

# getprotobyname

The Windows Sockets **getprotobyname** function retrieves the 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

The **getprotobyname** function returns a pointer to the [PROTOENT](#) structure containing the name(s) and protocol number that correspond to the protocol specified in the *name* parameter. All strings are null-terminated. The **PROTOENT** structure is allocated by the Windows Sockets library. An application must never attempt to modify this structure or to free any of its components. Furthermore, like [HOSTENT](#), only one copy of this structure is allocated per thread, so the application should copy any information that it needs before issuing any other Windows Sockets function calls.

## Return Values

If no error occurs, **getprotobyname** returns a pointer to the [PROTOENT](#). Otherwise, it returns a NULL pointer and a specific error number can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> 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 <i>name</i> parameter is not a valid part of the user address space.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[getprotobynumber](#), [WSAAsyncGetProtoByName](#)

## getprotobynumber

The Windows Sockets **getprotobynumber** function retrieves 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 **getprotobynumber** function returns a pointer to the [PROTOENT](#) structure as previously

described in [getprotobyname](#). The contents of the structure correspond to the given protocol number.

The pointer that is returned points to the structure 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, so the application should copy any information that it needs before issuing any other Windows Sockets function calls.

### Return Values

If no error occurs, **getprotobyname** returns a pointer to the [PROTOENT](#) structure. Otherwise, it returns a NULL pointer and a specific error number can be retrieved by calling [WSAGetLastError](#).

### Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> 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	A blocking Windows Socket 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[getprotobyname](#), [WSAAsyncGetProtoByNumber](#)

## getservbyname

The Windows Sockets **getservbyname** function retrieves 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 where *name* matches the **s\_name** member of the [SERVENT](#) structure or the **s\_aliases** member of the **SERVENT** structure. Otherwise, **getservbyname** matches both the *name* and the *proto*.

## Remarks

The **getservbyname** function returns a pointer to the **SERVENT** structure containing the name(s) and service number that match the string in the *name* parameter. All strings are null-terminated.

The pointer that is returned points to the **SERVENT** structure 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, so the application should copy any information it needs before issuing any other Windows Sockets function calls.

## Return Values

If no error occurs, **getservbyname** returns a pointer to the **SERVENT** structure. Otherwise, it returns a NULL pointer and a specific error number can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> 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	A blocking Windows Socket 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[getservbyport](#), [WSAAsyncGetServByName](#)

## getservbyport

The Windows Sockets **getservbyport** function retrieves 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** of the [SERVENT](#) structure. Otherwise, **getservbyport** matches both the *port* and the *proto* parameters.

### Remarks

The **getservbyport** function returns a pointer to a [SERVENT](#) structure as it does in the [getservbyname](#) function.

The **SERVENT** structure 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, so the application should copy any information it needs before issuing any other Windows Sockets function calls.

### Return Values

If no error occurs, **getservbyport** returns a pointer to the [SERVENT](#) structure. Otherwise, it returns a NULL pointer and a specific error number can be retrieved by calling [WSAGetLastError](#).

### Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> 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 <i>proto</i> parameter is not a valid part of the user address space.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[getservbyname](#), [WSAAsyncGetServByPort](#)

## GetService

**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..

The functions detailed in [Protocol-Independent Name Resolution](#) provide equivalent functionality in Windows Sockets 2.

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:

<b>Value</b>	<b>Name Space</b>
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 <i>dwNameSpace</i> .
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 can access a WINS server to perform the resolution.
NS_SAP	The Netware Service Advertising Protocol. This can 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 <b>lpComment</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_LOCALE	If this flag is set, the function stores data in the <b>lpLocale</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_DISPLAY_HINT	If this flag is set, the function stores data in the <b>dwDisplayHint</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_VERSION	If this flag is set, the function stores data in the <b>dwVersion</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_START_TIME	If this flag is set, the function stores data in the <b>dwTime</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_MACHINE	If this flag is set, the function stores data in the <b>lpMachineName</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_ADDRESSES	If this flag is set, the function stores data in the <b>lpServiceAddress</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_SD	If this flag is set, the function stores data in the <b>ServiceSpecificInfo</b> member of the data structures stored in <i>*lpBuffer</i> .
PROP_ALL	If this flag is set, the function stores data in all of the members of the data structures stored in <i>*lpBuffer</i> .

*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** can return one of the following extended error values:

Value	Meaning
<b>ERROR_INSUFFICIENT_BUFFER</b>	The buffer pointed to by <i>lpBuffer</i> 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 <i>*lpdwBufferSize</i> .
<b>ERROR_SERVICE_NOT_FOUND</b>	The specified service was not found, or the specified name space is not in use. The function return value is zero in this case.

### QuickInfo

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in nspapi.h.

**Import Library:** Link with wsock32.lib.

### See Also

[SetService](#), [NS\\_SERVICE\\_INFO](#), [SERVICE\\_INFO](#)

## getsockname

The Windows Sockets **getsockname** function retrieves 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*  
[in/out] The size of the *name* buffer.

## Remarks

The **getsockname** function retrieves the current name for the specified socket descriptor in *name*. It is used on the bound 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; the **getsockname** function provides the only way to determine the local association that has been set by the system.

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

The **getsockname** function does not always return information about the host address when the socket has been bound to an unspecified address, unless the socket has been connected with [connect](#) or [accept](#) (for example, using ADDR\_ANY). A Windows Sockets application must not assume that the address will be specified unless the socket is connected. The address that will be used for the socket is unknown unless the socket is connected when used in a multihomed host. If the socket is using a connectionless protocol, the address may not be available until I/O occurs on the socket.

## 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 <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The <i>name</i> or the <i>namelen</i> parameter is not a valid part of the user address space, or the <i>namelen</i> parameter 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 <a href="#">bind</a> , or ADDR_ANY is specified in <b>bind</b> but connection has not yet occurs.

## QuickInfo

**Windows NT:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with wsock32.lib.

## See Also

[bind](#), [getpeername](#), [socket](#)

# getsockopt

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 the Windows Sockets 2 Protocol-Specific Annex (a separate document included with the Platform SDK) for more information on 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

The **getsockopt** function 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 **LINGER** structure. 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 column 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 <a href="#">LINGER</a>	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 <a href="#">Windows Sockets 1.1 Blocking Routines &amp; EINPROGRESS</a> for a discussion of this topic.)
SO_PROTOCOL_INFO	<b>WSAPROTOCOL_INFO</b>	Description of protocol info for protocol that is bound to this socket.
SO_RCVBUF	int	Buffer size for receives
SO_REUSEADDR	BOOL	The socket can 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 <i>s</i> . 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 document.

### 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 option is reserved for future use with socket groups. This option is also exclusive to **getsockopt**. It indicates the identifier of the group to which this socket belongs. Socket group IDs are unique across all processes for a given service provider. If this socket is not a group socket, the value is NULL.

### SO\_GROUP\_PRIORITY

This option is reserved for future use with socket groups. Group priority indicates the priority of the specified socket relative to other sockets within the socket group. Values are non-negative integers, with zero corresponding to the highest priority. Priority values represent a hint to the underlying service provider about how potentially scarce resources should be allocated. For example, whenever two or more sockets are both ready to transmit

data, the highest priority socket (lowest value for `SO_GROUP_PRIORITY`) should be serviced first, with the remainder serviced in turn according to their relative priorities.

The `WSAENOPROTOOPT` error code is indicated for non group sockets or for service providers that 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 **LINGER** structure (pointed to by the *optval* parameter).

#### `SO_MAX_MSG_SIZE`

This is a get-only socket option that indicates the maximum outbound (send) 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. There is no provision to find out the maximum inbound message size

#### `SO_PROTOCOL_INFO`

This is a get-only option that supplies the **WSAPROTOCOL\_INFO** structure associated with this socket. See [WSAEnumProtocols](#) for more information about this structure.

#### `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 even if 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 that is already in use. On occasion, however, it can be necessary to "re-use" an address in this way. Because 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 that 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 `TCP_NODELAY` option is specific to TCP/IP service providers. 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. The process involves buffering send data when there is unacknowledged data already

"in flight" or buffering send data until a full-size packet can be sent. It is highly recommended that Windows Sockets implementations enable the Nagle Algorithm by default because, 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, and the time delays between the messages are maintained.

**Windows CE:** For IrSockets implementation:

- The `Af_irda.h` file must be explicitly included.
- The `WSAENETDOWN` return value is not supported.

IrSockets supports several special socket options:

Value	Type	Meaning
<code>IRLMP_ENUMDEVICES</code>	* <code>DEVICELIST</code>	Describes devices in range.
<code>IRLMP_IAS_QUERY</code>	* <code>IAS_QUERY</code>	Retrieve IAS attributes.
<code>IRLMP_SEND_PDU_LEN</code>	* <code>int</code>	Retrieves max number of bytes that can be sent in any one <code>send()</code> call while in <code>IRLMP_IRLPT_MODE</code> (printing). This value should be retrieved after the <code>connect()</code> completes but before any data is sent.

The **DEVICELIST** structure is an extendible array of device descriptions. IrSockets fills in as many device descriptions as can fit in the supplied buffer and returns in the `optlen` result parameter the required size if the buffer is of insufficient size. The device description consists of a device identifier necessary to form a `sockaddr_irda` structure and a displayable string describing the device.

The **IAS\_QUERY** structure is used to retrieve a single attribute of a single class. The application specifies the device and class to query and the attribute and attribute type. It is expected that the application allocates a buffer of the necessary size for the returned parameters.

Many SO level socket options are not meaningful to IrSockets. Only `SO_LINGER` and `SO_DONTLINGER` are specifically supported.

### 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 <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	One of the <i>optval</i> or the <i>optlen</i> parameters is not a valid part of the user address space, or the <i>optlen</i> parameter 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	The <i>level</i> parameter is unknown or invalid.
WSAENOPROTOOPT	The option is unknown or unsupported by the indicated protocol family.
WSAENOTSOCK	The descriptor is not a socket.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[setsockopt](#), [socket](#), [WSAAsyncSelect](#), [WSAConnect](#), [WSAGetLastError](#), [WSASetLastError](#)

## GetTypeByName

**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.

The functions detailed in [Protocol-Independent Name Resolution](#) provide equivalent functionality in Windows Sockets 2.

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

```

INT GetTypeByName(
    LPCTSTR 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** can return the following extended error value:

<b>Value</b>	<b>Meaning</b>
ERROR_SERVICE_DOES_NOT_EXIST	The specified service type is unknown.

**QuickInfo**

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in nspapi.h.

**Import Library:** Link with wsock32.lib.

**See Also**[GetNameByType](#)

# htonl

The Windows Sockets **htonl** function converts a **u\_long** from host to TCP/IP network byte order (which is big-endian).

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

**Parameters**

*hostlong*

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

**Remarks**

The **htonl** function 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

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

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[htons](#), [ntohl](#), [ntohs](#), [WSAHtonl](#), [WSAHtons](#), [WSANtohl](#), [WSANtohs](#)

# htons

The Windows Sockets **htons** function converts a **u\_short** from host to TCP/IP network byte order (which is big-endian).

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

## Parameters

*hostshort*

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

## Remarks

The **htons** function 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

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

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[htonl](#), [ntohl](#), [ntohs](#), [WSAHtonl](#), [WSAHtons](#), [WSANtohl](#), [WSANtohs](#)

## inet\_addr

The Windows Sockets **inet\_addr** function converts a string containing an (Ipv4) Internet Protocol dotted address into a proper address for the [IN\\_ADDR](#) structure.

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

### Parameters

*cp*

[in] A null-terminated character string representing a number expressed in the Internet standard "." (dotted) notation.

### Remarks

The **inet\_addr** 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. 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.

The parts that make up an address in "." notation can be decimal, octal or hexadecimal as specified in the C language. Numbers that start with "0x" or "0X" imply hexadecimal. Numbers that start with "0" imply octal. All other numbers are interpreted at decimal.

"4.3.2.16"	decimal
"004.003.002.020"	octal
"0x4.0x3.0x2.0x10"	hexidecimal
"4.003.002.0x10"	mix

**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 value containing a suitable binary representation of the Internet address given. If the string in the *cp* parameter 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.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[inet\\_ntoa](#)

## inet\_ntoa

The Windows Sockets **inet\_ntoa** function converts an (Ipv4) Internet network address into a string in Internet standard dotted format.

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

### Parameters

*in*

[in] A structure that represents an Internet host address.

### Remarks

The **inet\_ntoa** function takes an Internet address structure specified by the *in* parameter and returns an ASCII string representing the address in "." (dot) notation as in "a.b.c.d". The string

returned by **inet\_ntoa** resides in memory that 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. Therefore, the data should be copied before another Windows Sockets call is made.

## 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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[inet\\_addr](#)

# ioctlsocket

The Windows Sockets **ioctlsocket** function controls the I/O 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

The **ioctlsocket** function can be used on any socket in any state. It is used to set or retrieve operating parameters associated with the socket, independent of the protocol and communications subsystem. Here are the supported commands to use in the *cmd* parameter and their semantics:

### FIONBIO

Use with a nonzero *argp* parameter to enable the nonblocking mode of socket *s*. The *argp* parameter is zero if nonblocking is to be disabled. The *argp* parameter points to an

unsigned long value. When a socket is created, it operates in blocking mode by default (nonblocking mode is disabled). This is consistent with BSD sockets.

The [WSAAsyncSelect](#) and [WSAEventSelect](#) functions automatically set 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

Use to determine the amount of data pending in the network's input buffer that can be read from socket *s*. The *argp* parameter points to an unsigned long value in which [ioctlsocket](#) stores the result. If *s* is stream oriented (for example, type [SOCK\\_STREAM](#)), [FIONREAD](#) returns the amount of data that can be read in a single call to the [recv](#) function; this might 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

Use to determine whether or not all out-of-band data has been read. (See section [Windows Sockets 1.1 Blocking Routines & EINPROGRESS](#) for a discussion on Out of Band (OOB) data.) This applies only to a stream oriented socket (for example, type [SOCK\\_STREAM](#)) that has been configured for in-line reception of any out-of-band data ([SO\\_OOINLINE](#)). 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 data remains. If there is any normal data preceding the "urgent" (out of band) data, it will be received in order. (A [recv](#) or [recvfrom](#) will never mix out-of-band and normal data in the same call.) The *argp* parameter points to an unsigned long value in which [ioctlsocket](#) stores the boolean result.

### Compatibility

This [ioctlsocket](#) function performs only a subset of functions on a socket when compared to the [ioctl](#) function found in Berkeley sockets. The [ioctlsocket](#) function has no command parameter equivalent to the [FIOASYNC](#) of [ioctl](#), and [SIOCATMARK](#) is the only socket-level command that is supported by [ioctlsocket](#).

### 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

<a href="#">WSANOTINITIALISED</a>	A successful <a href="#">WSAStartup</a> must occur before using this function.
<a href="#">WSAENETDOWN</a>	The network subsystem has failed.
<a href="#">WSAEINPROGRESS</a>	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* parameter is not a valid part of the user address space.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[getsockopt](#), [setsockopt](#), [socket](#), [WSAAsyncSelect](#), [WSAEventSelect](#), [WSAIocctl](#)

# listen

The Windows Sockets **listen** function places a socket a state where it is listening 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 of the queue of pending connections. If this value is SOMAXCONN, then the underlying service provider responsible for socket *s* will set the backlog to a maximum "reasonable" value. There is no standard provision to find out the actual backlog value.

## Remarks

To accept connections, a socket is first created with the [socket](#) function and bound to a local address with the [bind](#) function, a backlog for incoming connections is specified with **listen**, and then the connections are accepted with the [accept](#) function. Sockets that are connection oriented, those of type SOCK\_STREAM for example, are used with **listen**. The socket *s* is put into "passive" mode where incoming connection requests are acknowledged and queued pending acceptance by the process.

The **listen** function is typically used by servers that can have more than one connection request at a time. If a connection request arrives and the queue is full, the client will receive an error with an indication of [WSAECONNREFUSED](#).

If there are no available socket descriptors, **listen** attempts to continue to function. 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.

**Windows CE:** As in 4.3BSD, illegal values (less than 1 or greater than 5) are replaced by the nearest valid value.

For IrSockets implementation:

- The Af\_irda.h file must be explicitly included.
- The WSAENETDOWN return value is not supported.
- The *backlog* parameter is currently limited (silently) to 2.

## 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. There is no standard provision to find out the actual backlog 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 <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEADDRINUSE	The socket's local address is already in use and the socket was not marked to allow address reuse with SO_REUSEADDR. This error usually occurs during execution of the <b>bind</b> function, but could be delayed until this function if the <b>bind</b> was to a partially wild-card address (involving ADDR_ANY) and if a specific address needs to be "committed" at the time of this function.
WSAEINPROGRESS	A blocking 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 <a href="#">bind</a> .
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.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[accept](#), [connect](#), [socket](#)

## ntohl

The Windows Sockets **ntohl** function converts a **u\_long** from TCP/IP network order to host byte order (which is big-endian).

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

### Parameters

*netlong*

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

### Remarks

The **ntohl** function takes a 32-bit number in TCP/IP network byte order and returns a 32-bit number in host byte order.

### Return Values

The **ntohl** function always returns a value in host byte order. If the *netlong* parameter was already in host byte order, then no operation is performed.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[htonl](#), [htons](#), [ntohs](#), [WSAHtonl](#), [WSAHtons](#), [WSANtohl](#), [WSANtohs](#)

## ntohs

The Windows Sockets **ntohs** function converts a **u\_short** from TCP/IP network byte order to host byte order (which is big-endian).

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

### Parameters

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

### Remarks

The **ntohs** function takes a 16-bit number in TCP/IP network byte order and returns a 16-bit number in host byte order.

### Return Values

The **ntohs** function returns the value in host byte order. If the *netshort* parameter was already in host byte order, then no operation is performed.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[htonl](#), [htons](#), [ntohl](#), [WSAHtonl](#), [WSAHtons](#), [WSANtohl](#), [WSANtohs](#)

## recv

The Windows Sockets **recv** function receives data from a connected 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] A flag specifying the way in which the call is made.

## Remarks

The **recv** function is used to read incoming data on connection-oriented sockets, or connectionless sockets. When using a connection-oriented protocol, the sockets must be connected before calling **recv**. When using a connectionless protocol, the sockets must be bound before calling **recv**.

The local address of the socket must be known. For server applications, use an explicit [bind](#) function or an implicit [accept](#) or [WSAAccept](#) function. Explicit binding is discouraged for client applications. For client applications the socket can become bound implicitly to a local address using [connect](#), [WSAConnect](#), [sendto](#), [WSASendTo](#), or [WSAJoinLeaf](#).

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

For connection-oriented sockets (type `SOCK_STREAM` for example), calling **recv** will return as much information as is currently available—up to the size of the buffer supplied. If the socket has been configured for in-line reception of out-of-band data (socket option `SO_OOINLINE`) and out-of-band data is yet unread, only out-of-band data will be returned. The application can use the **ioctlsocket** or **WSAIoctl** `SIOCATMARK` command to determine whether any more out-of-band data remains to be read.

For connectionless sockets (type `SOCK_DGRAM` or other message-oriented sockets), data is extracted from the first enqueued datagram (message) from the destination address specified by the [connect](#) function.

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. For TCP/IP, an application cannot receive from any multicast address until after becoming a group member.

If no incoming data is available at the socket, the **recv** call blocks and waits for data to arrive according to the blocking rules defined for [WSARecv](#) with the `MSG_PARTIAL` flag not set unless the socket is nonblocking. In this case, a value of `SOCKET_ERROR` is returned with the error code set to [WSAEWOULDBLOCK](#). The [select](#), [WSAAsyncSelect](#), or [WSAEventSelect](#) functions 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, and all data has been received, 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. The semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by or'ing 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. The function then returns the number of bytes currently pending to receive.
MSG_OOB	Process out-of-band data. (See section <a href="#">DECnet Out-Of-band data</a> 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 <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The <i>buf</i> parameter is not completely contained in a valid part of the user address space.
WSAENOTCONN	The socket is not connected.
WSAEINTR	The (blocking) call was canceled through <a href="#">WSACancelBlockingCall</a> .
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 <i>keep-alive</i> activity detecting a failure while the operation was in progress.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to <b>recv</b> on a socket after <a href="#">shutdown</a> has been invoked with <i>how</i> 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 <a href="#">bind</a> , or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled or (for byte stream sockets only) <i>len</i> 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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[recvfrom](#), [select](#), [send](#), [socket](#), [WSAAsyncSelect](#), [WSARecvEx](#)

# recvfrom

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] An indicator specifying the way in which the call is made.

*from*

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

*fromlen*

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

## Remarks

The **recvfrom** function reads incoming data on both connected and unconnected sockets and captures the address from which the data was sent. The socket must not be connected. The local address of the socket must be known. For server applications, this is usually done explicitly through **bind**. Explicit binding is discouraged for client applications. For client applications using this function, the socket can become bound implicitly to a local address through [sendto](#), [WSASendTo](#), or [WSAJoinLeaf](#).

For stream oriented sockets such as those of type SOCK\_STREAM, a call to **recvfrom** returns as much information as is currently available—up to the size of the buffer supplied. 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 yet unread, only out-of-band data will be returned. The application can use the [ioctlsocket](#) or [WSAIoctl](#) SIOCATMARK command to determine whether any more out-of-band data remains to be read. The *from* and *fromlen* parameters are ignored for connection-oriented sockets.

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

If the *from* parameter is nonzero and the socket is not connection oriented, (type SOCK\_DGRAM for example), the network address of the peer that sent the data is copied to the corresponding [SOCKADDR](#) structure. 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 in the [SOCKADDR](#) structure.

If no incoming data is available at the socket, the **recvfrom** function blocks and waits for data to arrive according to the blocking rules defined for [WSARecv](#) with the MSG\_PARTIAL flag not set unless the socket is 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, the call to **recvfrom** will complete immediately with zero bytes received. If the connection has been reset **recvfrom** 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. The semantics of this function are determined by the socket options and the *flags* parameter. The latter is constructed by or-ing the following values:

Value	Meaning
MSG_PEEK	Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue, and the function returns the number of bytes currently pending to receive.
MSG_OOB	Process out-of-band data. (See section <a href="#">DECnet Out-Of-band data</a> 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 <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The <i>buf</i> or <i>from</i> parameters are not part of the user address space, or the <i>fromlen</i> parameter is too small to accommodate the peer address.
WSAEINTR	The (blocking) call was canceled through <a href="#">WSACancelBlockingCall</a> .
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 <a href="#">bind</a> , 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) <i>len</i> was zero or negative.
WSAEISCONN	The socket is connected. This function is not permitted with a connected socket, whether the socket is connection-oriented or connectionless.
WSAENETRESET	The connection has been broken due to the "keep-alive" activity detecting a failure while the operation was in progress.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only send operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to <b>recvfrom</b> on a socket after <a href="#">shutdown</a> has been invoked with <i>how</i> set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the <b>recvfrom</b> operation would block.

WSAEMSGSIZE	The message was too large to fit into the specified buffer and was truncated.
WSAETIMEDOUT	The connection has been dropped, because of a network failure or because the system on the other end went down without notice.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a "hard" or "abortive" close. The application should close the socket as it is no longer usable. On a UDP datagram socket this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[recv](#), [send](#), [socket](#), [WSAAsyncSelect](#), [WSAEventSelect](#)

# select

The Windows Sockets **select** function determines the status of one or more sockets, waiting if necessary, to perform synchronous I/O.

```
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 parameter is ignored; it is included only for compatibility with Berkeley sockets.

*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

The **select** 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. For the purpose of this restriction, sockets are considered to be from the same service provider if the **WSAPROTOCOL\_INFO** structures describing their protocols have the same *providerId* value. Upon return, the structures are updated to reflect the subset of these sockets that meet the specified condition. The **select** function returns the number of sockets meeting the conditions. A set of macros is provided for manipulating an **FD\_SET** structure. These macros are compatible with those used in the Berkeley software, but the underlying representation is completely different.

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

For connection-oriented sockets, readability can also indicate that a request to close the socket has been received from the peer. If the virtual circuit was closed gracefully, and all data was received, 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 *writfds* identifies the sockets that are to be checked for writability. If a socket is processing a **connect** call (nonblocking), a socket is writable if the connection establishment successfully completes. If the socket is not processing a **connect** call, writability means a **send**, **sendto**, or **WSASendto** are guaranteed to succeed. However, they can block on a blocking socket if the *len* parameter 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 the sockets that are to be checked for the presence of out-of-band data (see section **DECnet Out-Of-band data** for a discussion of this topic) or any exceptional error conditions.

**Important** Out-of-band data will only be reported in this way if the option **SO\_OOBINLINE** is **FALSE**. If a socket is processing a **connect** call (nonblocking), failure of the connect attempt is indicated in *exceptfds* (application must then call **getsockopt** **SO\_ERROR** to determine the error value to describe why the failure occurred). This document does not define which other errors will be included.

Any two of the parameters, *readfds*, *writfds*, or *exceptfds*, can be given as **NULL**. At least one must be non-**NULL**, and any non-**NULL** descriptor set must contain at least one handle to a socket.

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

*readfds:*

- If **listen** has been called and 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

*writelfds:*

- If processing a **connect** call (nonblocking), connection has succeeded
- Data can be sent

*exceptfds:*

- If processing a **connect** call (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 an **FD\_SET** structure 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 **TIMEVAL** structure that specifies the maximum time that **select** should wait before returning. When **select** returns, the contents of the **TIMEVAL** structure are not altered. If **TIMEVAL** is initialized to {0, 0}, **select** will return immediately; this is used to "poll" the state of the selected sockets. If **select** returns immediately, 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.

**Note** The **select** function has no effect on the persistence of socket events registered with [WSAAsyncSelect](#) or [WSAEventSelect](#).

**Return Values**

The **select** function returns the total number of socket handles that 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.

## Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> must occur before using this function.
WSAEFAULT	The Windows Sockets implementation was unable to allocate needed resources for its internal operations, or the <i>readfds</i> , <i>writesfds</i> , <i>exceptfds</i> , or <i>timeval</i> parameters are not part of the user address space.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	The <i>timeout</i> value is not valid, or all three descriptor parameters were NULL.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .
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 that is not a socket.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[accept](#), [connect](#), [recv](#), [recvfrom](#), [send](#), [WSAAsyncSelect](#), [WSAEventSelect](#)

# send

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] An indicator specifying the way in which the call is made.

**Remarks**

The **send** function 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 using [getsockopt](#) to retrieve 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 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 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 client and server machines. The [select](#), [WSAAsyncSelect](#) or [WSAEventSelect](#) functions can be used to determine when it is possible to send more data.

Calling **send** with a zero *len* parameter is permissible and will be treated by implementations as successful. In such cases, **send** will return zero as a valid value. For message-oriented sockets, a zero-length transport datagram is sent.

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

<b>Value</b>	<b>Meaning</b>
<code>MSG_DONTROUTE</code>	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag.
<code>MSG_OOB</code>	Send out-of-band data (stream-style socket such as <code>SOCK_STREAM</code> only. Also see <a href="#">DECnet Out-Of-band data</a> for a discussion of this topic).

**Windows CE:** For IrSockets implementation, the `Af_irda.h` file must be explicitly included.

**Return Values**

If no error occurs, **send** returns the total number of bytes sent, which can be less than the number indicated by *len* for nonblocking sockets. Otherwise, a value of `SOCKET_ERROR` is returned, and a specific error code can be retrieved by calling [WSAGetLastError](#).

**Error Codes**

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> 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. Call <a href="#">setsockopt</a> with the SO_BROADCAST parameter to allow the use of the broadcast address.
WSAEINTR	A blocking Windows Sockets 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The <i>buf</i> parameter is not completely contained in a valid part of the user address space.
WSAENETRESET	The connection has been broken due to the "keep-alive" activity detecting a failure while the operation was in progress.
WSAENOBUFS	No buffer space is available.
WSAENOTCONN	The socket is not connected.
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	MSG_OOB was specified, but the socket is not stream-style such as type SOCK_STREAM, out-of-band data is not supported in the communication domain associated with this socket, or the socket is unidirectional and supports only receive operations.
WSAESHUTDOWN	The socket has been shut down; it is not possible to <b>send</b> on a socket after <a href="#">shutdown</a> 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 <a href="#">bind</a> , or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOINLINE 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 UDP 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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[recv](#), [recvfrom](#), [select](#), [sendto](#), [socket](#), [WSAAsyncSelect](#), [WSAEventSelect](#)

# sendto

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 (possibly connected) socket.

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

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

*flags*  
[in] An indicator specifying 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

The **sendto** function 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 using [getsockopt](#) to retrieve 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 used [setsockopt](#) with `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 then marked as bound. An application can use [getsockname](#) to determine the local socket name in this case.

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

The **sendto** function 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 connected to a specific address, the *to* parameter overrides the destination address for that particular datagram only. On a connection-oriented socket, the *to* and *toLen* parameters are ignored, making **sendto** equivalent to [send](#).

For sockets using IP (version 4):

To send a broadcast (on a `SOCK_DGRAM` only), the address in the *to* parameter should be constructed using the special IP address `INADDR_BROADCAST` (defined in `WINSOCK2.H`), together with the intended port number. It is generally inadvisable for a broadcast datagram to exceed the size at which fragmentation 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 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 client and server systems. The [select](#), [WSAAsyncSelect](#) or [WSAEventSelect](#) function can be used to determine when it is possible to send more data.

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

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

Value	Meaning
<code>MSG_DONTROUTE</code>	Specifies that the data should not be subject to routing. A Windows Sockets service provider can choose to ignore this flag.
<code>MSG_OOB</code>	Send out-of-band data (stream-style socket such as <code>SOCK_STREAM</code> only. Also see <a href="#">DECnet Out-Of-band data</a> for a discussion of this topic.)

## Return Values

If no error occurs, **sendto** returns the total number of bytes sent, which 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 <a href="#">WSAStartup</a> 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. Call <a href="#">setsockopt</a> with the <code>SO_BROADCAST</code> parameter to allow the use of the broadcast address.
WSAEINVAL	An unknown flag was specified, or <code>MSG_OOB</code> was specified for a socket with <code>SO_OOBINLINE</code> enabled.
WSAEINTR	A blocking Windows Sockets 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The <i>buf</i> or <i>to</i> parameters are not part of the user address space, or the <i>toLen</i> parameter is too small.
WSAENETRESET	The connection has been broken due to "keep-alive" activity detecting a failure while the operation was in progress.
WSAENOBUFS	No buffer space is available.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only)
WSAENOTSOCK	The descriptor is not a socket.
WSAEOPNOTSUPP	<code>MSG_OOB</code> was specified, but the socket is not stream-style such as type <code>SOCK_STREAM</code> , 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 <b>sendto</b> on a socket after <a href="#">shutdown</a> has been invoked with <i>how</i> set to <code>SD_SEND</code> or <code>SD_BOTH</code> .
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 UDP 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 remote address is not a valid address, for example, ADDR_ANY.
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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[recv](#), [recvfrom](#), [select](#), [send](#), [socket](#), [WSAAsyncSelect](#), [WSAEventSelect](#)

# SetService

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

The functions detailed in [Protocol-Independent Name Resolution](#) provide equivalent functionality in Windows Sockets 2.

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*

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 <i>dwNameSpace</i> .
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\hosts 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:

<b>Value</b>	<b>Meaning</b>
SERVICE_REGISTER	Register the network service with the name space. This operation can 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 <b>ServiceSpecificInfo</b> member of the <a href="#">SERVICE_INFO</a> structure pointed to by <i>lpServiceInfo</i> to pass a <a href="#">SERVICE_TYPE_INFO_ABS</a> structure. You must also set the <b>ServiceType</b> member of the <b>SERVICE_INFO</b> structure. Other <b>SERVICE_INFO</b> 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:

<b>Value</b>	<b>Name Space</b>
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** can return the following extended error value:

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

**QuickInfo**

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in nspapi.h.

**Import Library:** Link with wsock32.lib.

**See Also**

[GetService](#), [SERVICE\\_INFO](#), [SERVICE\\_TYPE\\_INFO\\_ABS](#)

## setsockopt

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 *levels* include SOL\_SOCKET and IPPROTO\_TCP. See the Windows Sockets 2 Protocol-Specific Annex (a separate document included with the Platform SDK) for more information on protocol-specific levels.

*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

The **setsockopt** function 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 (OOB data for example) is received in the normal data stream, and 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 that 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. The *optlen* parameter should be equal to **sizeof(int)** for Boolean options. For other options, *optval* points to the an integer or structure that contains the desired value for the option, and *optlen* is the length of the integer or structure.

The following options are supported for **setsockopt**. For default values of these options, see the description. 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 <i>l_onoff</i> set to zero.
SO_DONTROUTE	BOOL	Do not route: send directly to interface.
SO_GROUP_PRIORITY	int	Reserved for future use with socket groups. Specify the relative priority to be established for sockets that are part of a socket group.
SO_KEEPAIVE	BOOL	Send keepalives
SO_LINGER	struct <a href="#">LINGER</a>	Linger on close if unsent data is present.
SO_OOINLINE	BOOL	Receive out-of-band data in the normal data stream. (See section <a href="#">DECnet Out-Of-band data</a> for a discussion of this topic.)

SO_RCVBUF	int	Specify the total per-socket buffer space reserved for receives. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.
SO_REUSEADDR	BOOL	Allow the socket to be bound to an address that is already in use. (See <a href="#">bind.</a> )
SO_SNDBUF	int	Specify the total per-socket buffer space reserved for sends. This is unrelated to SO_MAX_MSG_SIZE or the size of a TCP window.
PVD_CONFIG	Service Provider Dependent	This object stores the configuration information for the service provider associated with socket <i>s</i> . The exact format of this data structure is service provider specific.

### *level* = IPPROTO\_TCP1

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 (available in Microsoft implementation of Windows Sockets 2)
SO_SNDLOWAT	int	Send low water mark
SO_SNDTIMEO	int	Send time-out (available in Microsoft implementation of Windows Sockets 2)
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 document.

### SO\_GROUP\_PRIORITY

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

with the remainder serviced in turn according to their relative priorities.

The [WSAENOPROTOOPT](#) error is indicated for nongroup sockets or for service providers that 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

The SO\_LINGER option 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 [LINGER](#) structure (pointed to by the *optval* parameter) with these members **l\_onoff** and **l\_linger** set appropriately.

#### SO\_REUSEADDR

By default, a socket cannot be bound (see [bind](#)) to a local address that is already in use. On occasion, however, it can 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**. The option is interpreted only at the time of the [bind](#). It is therefore unnecessary and harmless to set the option on a socket that is not to be bound to an existing address. Setting or resetting the option after the **bind** has no effect on this or any other socket.

#### SO\_RCVBUF and 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 even when 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 the socket specified in the *s* parameter. The exact format of this data structure is specific to each service provider.

#### TCP\_NODELAY

The TCP\_NODELAY option is specific to TCP/IP service providers. The Nagle algorithm is disabled if the TCP\_NODELAY option is enabled (and vice versa). The process involves buffering send data when there is unacknowledged data already "in flight" or buffering send data 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, and the time delays between the messages are maintained. Application writers should not set TCP\_NODELAY unless the impact of doing so is well-understood and desired because setting TCP\_NODELAY can have a significant negative impact on network and application performance.

**Windows CE:** The `SO_RCVBUF` option is not supported. If you attempt to use this option `setsockopt` returns `WSAEOPNOTSUPP`.

To set the socket to secure mode, the option level parameter, *level*, must be set to `SO_SOCKET`, the option name, *optname* to `SO_SECURE`, and the option value, *optval*, must be a pointer to a `DWORD` containing `SO_SEC_SSL`. These settings ensure that the Unified Secure Sockets Layer (SSL) package be used. For example,

```
DWORD optval = SO_SEC_SSL;
err = setsockopt(
    Socket,
    SOL_SOCKET,
    SO_SECURE,
    &optval,
    sizeof(optval)
);
```

In addition to the normal error values, the `setsockopt` function can return an additional error code, namely, `WSAEISCONN`, to signify that the socket can not switch to secure mode once it has been connected.

When used in the context of SSL, the `WSAENOPROTOOP` error code acquires additional meaning, to indicate that the option level does not equal to `SO_SOCKET`.

For IrSocket implementation, `Af_irda.h` must be explicitly included.

The `WSAENETDOWN` return value is not supported for IrSockets.

IrSockets provides two settable socket options:

Value	Type	Meaning
<code>IRLMP_IAS_SET</code>	* <code>IAS_SET</code>	Sets IAS attributes.
<code>IRLMP_IRLPT_MODE</code>	* <code>int</code>	In non-zero, enables IrLPT mode for printing to IrDA printers.

The `IRLMP_IAS_SET` socket option allows the application to set a single attribute of a single class in the local IAS. The application specifies the class to set and the attribute and attribute type. It is expected that the application allocate a buffer of the necessary size for the passed parameters.

The `IRLMP_RAW_MODE` socket option allows the application to switch between TinyTP mode and unreliable IrLMP mode. If it is not set, IrSockets are assumed to use TinyTP. This option is only available after issuing the socket function and before issuing any other Windows Sockets function.

Many `SO_` level socket options are not meaningful to IrSockets. Only `SO_LINGER` is specifically supported.

## 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 <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	<i>optval</i> is not in a valid part of the process address space or <i>optlen</i> parameter 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	<i>level</i> is not valid, or the information in <i>optval</i> is not valid.
WSAENETRESET	Connection has timed out when SO_KEEPALIVE is set.
WSAENOPROTOOPT	The option is unknown or unsupported for the specified provider or socket (see SO_GROUP_PRIORITY limitations).
WSAENOTCONN	Connection has been reset when SO_KEEPALIVE is set.
WSAENOTSOCK	The descriptor is not a socket.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[bind](#), [getsockopt](#), [ioctlsocket](#), [socket](#), [WSAAsyncSelect](#), [WSAEventSelect](#)

# shutdown

The Windows Sockets **shutdown** function disables sends 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

The **shutdown** function is used on all types of sockets to disable reception, transmission, or both.

If the *how* parameter is SD\_RECEIVE, subsequent calls to the [recv](#) function 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 the *how* parameter is SD\_SEND, subsequent calls to the [send](#) function are disallowed. For TCP sockets, a FIN will be sent after all data is sent and acknowledged by the receiver.

Setting *how* to SD\_BOTH disables both sends and receives as described above.

The **shutdown** function does not close the socket. Any 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:

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.
4. Call [closesocket](#).

**Note** The **shutdown** function 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 a socket that has been shutdown.

## 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 <b>WSAStartup</b> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEINVAL	The <i>how</i> parameter 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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[connect](#), [socket](#)

# socket

The Windows Sockets **socket** function creates a socket that 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.

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 and don't need to be

specified now because an application can dynamically discover the attributes of each available transport protocol through the [WSAEnumProtocols](#) function. Socket type definitions will appear in WINSOCK2.H, which will be periodically updated as new socket types, address families and protocols are defined.

*protocol*

[in] A particular protocol to be used with the socket that 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. The socket that is created will have the overlapped attribute as a default. For Microsoft operating systems, the Microsoft-specific socket option, SO\_OPENTYPE, defined in MSWSOCK.H can affect this default. See Microsoft-specific documentation for a detailed description of SO\_OPENTYPE. Sockets without the overlapped attribute can be created by using [WSASocket](#). All functions that allow overlapped operation ([WSASend](#), [WSARecv](#), [WSASendTo](#), [WSARecvFrom](#), and [WSAIocctl](#)) also support non-overlapped usage on an overlapped socket if the values for parameters related to overlapped operation are NULL.

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

**Important** 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.

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 connected 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.

**Windows CE:** Windows CE supports the PF\_INET and AF\_IRDA ARPA Internet address formats.

To use IrSock, set the *af* parameter to AF\_IRDA and set the *protocol* parameter to NULL.

Windows CE supports SOCK\_STREAM and SOCK\_DGRAM socket types.

### 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 <b>WSAStartup</b> 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.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[accept](#), [bind](#), [connect](#), [getsockname](#), [getsockopt](#), [ioctlsocket](#), [listen](#), [recv](#), [recvfrom](#), [select](#), [send](#), [sendto](#), [setsockopt](#), [shutdown](#), [WSASocket](#)

## TransmitFile

**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 (asynchronous) 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 **TransmitFile** function succeeds, the return value is TRUE. Otherwise, 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.

## QuickInfo

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in mswsock.h.

**Import Library:** Link with mswsock.lib.

# WSAAccept

The Windows Sockets **WSAAccept** function conditionally accepts a connection based on the return value of a condition function, optionally creates or joins a socket group, provides QOS flowspecs, and allows transfer of connection data.

```

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

```

## Parameters

*s*

[in] A descriptor identifying a socket that is listening for connections after a call to the [listen](#) function.

*addr*

[out] An optional pointer to a buffer that receives 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 established when the socket was created.

*addrlen*

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

*lpfnCondition*

[in] The procedure instance address of the optional, application-supplied condition function that will make an accept/reject decision based on the caller information passed in as parameters, and optionally create 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 the value of the *dwCallbackData* parameter of the condition function. This parameter is not interpreted by Windows Sockets.

## Remarks

The [WSAAccept](#) function extracts the first connection on the queue of pending connections on socket *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, **WSAAccept** 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 socket *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, **WSAAccept** rejects the connection request. The condition function runs in the same thread as this function 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.

A socket in the default mode (blocking) will block until a connection is present when an application calls **WSAAccept** and no connections are pending on the queue.

A socket in the nonblocking mode (blocking) fails with the error WSAEWOULDBLOCK when an application calls **WSAAccept** and no connections are pending on the queue. After **WSAAccept** succeeds and returns a new socket handle, the accepted socket cannot be used to accept any more connections. The original socket remains open and listens for new connection requests.

The *addr* parameter 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.

A 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  
);
```

The **ConditionFunc** is a placeholder for the application-supplied callback function. The actual condition function must reside in a DLL or application module. It is exported in the module definition file. Use [MakeProcInstance](#) to get a procedure-instance address for the callback function.

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

The *lpSQOS* parameter references the **FLOWSPEC** structures 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 indicates that there is no caller supplied QOS and that no negotiation is possible. A non-`NULL` *lpSQOS* pointer indicates that a QOS negotiation is to occur or that the provider is prepared to accept the QOS request without negotiation.

The *lpGQOS* parameter (reserved for future use with socket groups) references the **FLOWSPEC** structures 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 quality of service. quality of service information can be returned if negotiation is to occur.

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

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

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

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.

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

## Return Values

If no error occurs, **WSAAccept** returns a value of type *SOCKET* that 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 <b>WSAStartup</b> <i>must</i> 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 <i>addr</i> parameter is too small or the <i>addr</i> or <i>lpszCondition</i> are not part of the user address space.
WSAEINTR	A blocking Windows Sockets 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress.
WSAEINVAL	<b>listen</b> was not invoked prior to <b>WSAAccept</b> , parameter <i>g</i> 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 <i>g</i> 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 <b>WSAAccept</b> 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.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[accept](#), [bind](#), [connect](#), [getsockopt](#), [listen](#), [select](#), [socket](#), [WSAAsyncSelect](#), [WSAConnect](#)

# WSAAddressToString

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,  
    IN OUT LPDWORD lpdwAddressStringLength  
);
```

## Parameters

*lpsaAddress*

[in] A pointer to the [SOCKADDR](#) structure to translate into a string.

*dwAddressLength*

[in] The length of the address in **SOCKADDR**, which may vary in size with different protocols.

*lpProtocolInfo*

[in] (Optional) The [WSAPROTOCOL\\_INFO](#) structure for a particular provider. If this is NULL, the call is routed to the provider of the first protocol supporting the address family indicated in *lpsaAddress*.

*lpszAddressString*

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

*lpdwAddressStringLength*

[in/out] On input, the length of the *AddressString* buffer. On output, returns the length of the string actually copied into the buffer. If the supplied buffer is not large enough, the function fails with a specific error of **WSAEFAULT** and this parameter is updated with the required size in bytes.

## Return Values

If no error occurs, **WSAAddressToString** returns a value of zero. Otherwise, the value **SOCKET\_ERROR** is returned, and a specific error number can be retrieved by calling [WSAGetLastError](#).

## Error Codes

**WSAEFAULT**

The specified *lpcsAddress*, *lpProtocolInfo*, *lpszAddressString* are not all in the address space of the process, or the *lpszAddressString* buffer is too small. Pass in a larger buffer.

WSAEINVAL	The specified address is not a valid socket address, or there was no transport provider supporting its indicated address family.
WSANOTINITIALIZED	The Winsock 2 DLL has not been initialized. The application <i>must</i> first call <a href="#">WSAStartup</a> before calling any Windows Sockets functions.
WSA NOT ENOUGH MEMORY	There was insufficient memory to perform the operation.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

# WSAAsyncGetHostByAddr

The Windows Sockets **WSAAsyncGetHostByAddr** function asynchronously retrieves host information that corresponds to an address.

```
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 that will 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. The data area *must* be larger than the size of a **HOSTENT** structure because the supplied data area is used by Windows

Sockets to contain a **HOSTENT** structure and all of the data referenced by members of the **HOSTENT** structure. A buffer of MAXGETHOSTSTRUCT bytes is recommended.

*buflen*

[in] The size of data area for the *buf* parameter.

## Remarks

The **WSAAsyncGetHostByAddr** function is an asynchronous version of [gethostbyaddr](#). It is used to retrieve the host name and address information that corresponds to a network address. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that 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 has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter 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 members of this structure, the original buffer address is cast to a [HOSTENT](#) structure pointer and accessed as appropriate.

If the error code is WSAENOBUFFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting 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 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 WSAENOBUFFS error, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in WINSOCK2.H).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLLEN, defined in WINSOCK2.H as:

```
#define WSAGETASYNCERROR(lParam)          HIWORD(lParam)
#define WSAGETASYNCBUFLLEN(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. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetHostByAddr** returns a nonzero value of type HANDLE that 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](#), or it can be used to match up asynchronous operations and completion messages by examining the *wParam* message parameter.

If the asynchronous operation could not be initiated, **WSAAsyncGetHostByAddr** returns a zero value, and a specific error number can be retrieved by calling [WSAGetLastError](#).

## 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	<i>addr</i> or <i>buf</i> 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 <a href="#">WSAStartup</a> 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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[gethostbyaddr](#), [HOSTENT](#), [WSACancelAsyncRequest](#)

# WSAAsyncGetHostByName

The Windows Sockets **WSAAsyncGetHostByName** function asynchronously retrieves host information corresponding to a host name.

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

## Parameters

*hWnd*

[in] The handle of the window that will 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. The data area must be larger than the size of a **HOSTENT** structure because the supplied data area is used by Windows Sockets to contain a **HOSTENT** structure and all of the data referenced by members of the **HOSTENT** structure. A buffer of MAXGETHOSTSTRUCT bytes is recommended.

*buflen*

[in] The size of data area for the *buf* parameter.

## Remarks

The **WSAAsyncGetHostByName** function is an asynchronous version of [gethostbyname](#), and is used to retrieve host name and address information corresponding to a host name. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque "asynchronous task handle" that 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 has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter 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.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting 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 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 WSAENOBUFFS error, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in WINSOCK2.H).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLLEN, defined in WINSOCK2.H as:

```
#define WSAGETASYNCERROR(lParam)          HIWORD(lParam)
#define WSAGETASYNCBUFLLEN(lParam)      LOWORD(lParam)
```

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

**WSAAsyncGetHostName** 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. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetHostName** returns a nonzero value of type HANDLE that 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**, or it can be used to match up asynchronous operations and completion messages by examining the *wParam* message parameter.

If the asynchronous operation could not be initiated, **WSAAsyncGetHostName** returns a zero value, and a specific error number can be retrieved by calling **WSAGetLastError**.

### 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.
WSAENOBUFFS	Insufficient buffer space is available.
WSAEFAULT	The <i>name</i> or <i>buf</i> parameter 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 <a href="#">WSAStartup</a> 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.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[gethostbyname](#), [HOSTENT](#), [WSACancelAsyncRequest](#)

## WSAAsyncGetProtoByName

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

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

### Parameters

*hWnd*

[in] The handle of the window that will 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. The data area must be larger than the size of a **PROTOENT** structure because the data area is used by Windows Sockets to contain a **PROTOENT** structure and all of the data that is referenced by members of the **PROTOENT** structure. A buffer of MAXGETHOSTSTRUCT bytes is recommended.

*buflen*

[out] The size of data area for the *buf* parameter.

## Remarks

The **WSAAsyncGetProtoByName** function is an asynchronous version of [getprotobyname](#). It is used to retrieve the protocol name and number from the Windows Sockets database corresponding to given protocol name. Windows Sockets initiates the operation and returns to the caller immediately, passing back an opaque, asynchronous task handle that 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 has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter 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 members of this structure, the original buffer address should be cast to a **PROTOENT** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting 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 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).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLLEN, defined in WINSOCK2.H as:

```
#define WSAGETASYNCERROR(lParam)          HIWORD(lParam)
#define WSAGETASYNCBUFLLEN(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. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetProtoByName** returns a nonzero value of type HANDLE that 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](#), or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

If the asynchronous operation could not be initiated, **WSAAsyncGetProtoByName** returns a zero value, and a specific error number can be retrieved by calling [WSAGetLastError](#).

## 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	The <i>name</i> or <i>buf</i> parameter 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 <a href="#">WSAStartup</a> 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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[getprotobyname](#), [WSACancelAsyncRequest](#)

## WSAAsyncGetProtoByNumber

The Windows Sockets **WSAAsyncGetProtoByNumber** function asynchronously retrieves protocol information corresponding to a protocol number.

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

### Parameters

*hWnd*

[in] The handle of the window that will 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. The data area must be larger than the size of a **PROTOENT** structure because the data area is used by Windows Sockets to contain a **PROTOENT** structure and all of the data that is referenced by members of the **PROTOENT** structure. A buffer of MAXGETHOSTSTRUCT bytes is recommended.

*buflen*

[in] The size of data area for the *buf* parameter.

### Remarks

The **WSAAsyncGetProtoByNumber** 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 that 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 has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter 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 members of this structure, the original buffer address is cast to a **PROTOENT** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting 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 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).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLLEN, defined in WINSOCK2.H as:

```
#define WSAGETASYNCERROR(lParam)          HIWORD(lParam)
#define WSAGETASYNCBUFLLEN(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. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetProtoByNumber** returns a nonzero value of type HANDLE that 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](#), or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

If the asynchronous operation could not be initiated, **WSAAsyncGetProtoByNumber** returns a zero value, and a specific error number can be retrieved by calling [WSAGetLastError](#).

## 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	The <i>buf</i> parameter 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 <a href="#">WSAStartup</a> 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.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[getprotobynumber](#), [WSACancelAsyncRequest](#)

## WSAAsyncGetServByName

The Windows Sockets **WSAAsyncGetServByName** function asynchronously retrieves service information corresponding to a service name and port.

```
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 that 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. The data area must be larger than the size of a **SERVENT** structure because the data area supplied is used by Windows Sockets to contain a **SERVENT** structure and all of the data that is referenced by members of the **SERVENT** structure. A buffer of MAXGETHOSTSTRUCT bytes is recommended.

*buflen*

[in] The size of data area for the *buf* parameter.

## Remarks

The **WSAAsyncGetServByName** 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 that 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 has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter 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 members of this structure, the original buffer address should be cast to a **SERVENT** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting 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 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 WSAENOBUFFS error, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in WINSOCK2.H).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLLEN, defined in WINSOCK2.H as:

```
#define WSAGETASYNCERROR(lParam)          HIWORD(lParam)
#define WSAGETASYNCBUFLLEN(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. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetServByName** returns a nonzero value of type `HANDLE` that 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](#), or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

If the asynchronous operation could not be initiated, **WSAAsyncServByName** returns a zero value, and a specific error number can be retrieved by calling [WSAGetLastError](#).

### 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.

<code>WSAENETDOWN</code>	The network subsystem has failed.
<code>WSAENOBUFS</code>	Insufficient buffer space is available.
<code>WSAEFAULT</code>	<i>buf</i> is not in a valid part of the process address space.
<code>WSAHOST_NOT_FOUND</code>	Authoritative Answer Host not found.
<code>WSATRY_AGAIN</code>	Non-Authoritative Service not found, or server failure.
<code>WSANO_RECOVERY</code>	Nonrecoverable errors, the services database is not accessible.
<code>WSANO_DATA</code>	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.

<code>WSANOTINITIALISED</code>	A successful <a href="#">WSAStartup</a> must occur before using this function.
<code>WSAENETDOWN</code>	The network subsystem has failed.
<code>WSAEINPROGRESS</code>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<code>WSAEWOULDBLOCK</code>	The asynchronous operation cannot be scheduled at this time due to resource or other constraints within the Windows Sockets implementation.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes  
**Windows CE:** Unsupported.  
**Header:** Declared in winsock2.h.  
**Import Library:** Link with ws2\_32.lib.

### See Also

[getservbyname](#), [WSACancelAsyncRequest](#)

## WSAAsyncGetServByPort

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

```
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 that 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] pointer to the data area to receive the [SERVENT](#) data. The data area must be larger than the size of a **SERVENT** structure because the data area supplied is used by Windows Sockets to contain a **SERVENT** structure and all of the data that is referenced by members of the **SERVENT** structure. A buffer of MAXGETHOSTSTRUCT bytes is recommended.

*buflen*

[in] The size of data area for the *buf* parameter.

### Remarks

The **WSAAsyncGetServByPort** 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 that 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 has completed, the application window indicated by the *hWnd* parameter receives message in the *wMsg* parameter. The *wParam* parameter 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 members of this structure, the original buffer address should be cast to a **SERVENT** structure pointer and accessed as appropriate.

If the error code is WSAENOBUFS, the size of the buffer specified by *buflen* in the original call was too small to contain all the resulting 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 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, the application should provide a buffer of at least MAXGETHOSTSTRUCT bytes (as defined in WINSOCK2.H).

The error code and buffer length should be extracted from the *lParam* using the macros WSAGETASYNCERROR and WSAGETASYNCBUFLLEN, defined in WINSOCK2.H as:

```
#define WSAGETASYNCERROR(lParam)          HIWORD(lParam)
#define WSAGETASYNCBUFLLEN(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. It does not imply success or failure of the operation itself.

If no error occurs, **WSAAsyncGetServByPort** returns a nonzero value of type HANDLE that 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](#), or it can be used to match up asynchronous operations and completion messages, by examining the *wParam* message parameter.

If the asynchronous operation could not be initiated, **WSAAsyncGetServByPort** returns a zero value, and a specific error number can be retrieved by calling [WSAGetLastError](#).

## 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	<i>proto</i> or <i>buf</i> 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 <a href="#">WSAStartup</a> 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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[getservbyport](#), [WSACancelAsyncRequest](#)

# WSAAsyncSelect

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 that will receive a message when a network event occurs.

*wMsg*

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

*lEvent*

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

## Remarks

The **WSAAsyncSelect** function is used to request that WS2\_32.DLL should send a message to the window *hWnd* whenever it detects any of the network events specified by the *lEvent* parameter. The message that should be sent is specified by the *wMsg* parameter. The socket for which notification is required is identified by the *s* parameter.

The **WSAAsyncSelect** function automatically sets socket *s* to nonblocking mode, regardless of the value of *lEvent*. See the [ioctlsocket](#) functions for information on how to set the nonblocking socket back to blocking mode.

The *lEvent* parameter is constructed by using the bitwise OR operator with any of the values specified in the following list.

Value	Meaning
FD_READ	Want to receive notification of readiness for reading
FD_WRITE	Want to receive notification of readiness for writing
FD_OOB	Want to receive notification of the arrival of out-of-band data
FD_ACCEPT	Want to receive notification of incoming connections
FD_CONNECT	Want to receive notification of completed connection or multi-point <i>join</i> operation
FD_CLOSE	Want to receive notification of socket closure
FD_QOS	Want to receive notification of socket Quality of Service (QOS) changes
FD_GROUP_QOS	Want to receive notification of socket group Quality of Service (QOS) changes (reserved for future use with socket groups)
FD_ROUTING _INTERFACE_CHANGE	Want to receive notification of routing interface changes for the specified destination(s)
FD_ADDRESS_LIST _CHANGE	Want to receive notification of local address list changes for the socket's protocol family

Issuing a **WSAAsyncSelect** for a socket cancels any previous **WSAAsyncSelect** or [WSAEventSelect](#) for the same socket. For example, to receive notification for both reading and

writing, the application must call **WSAAsyncSelect** with both `FD_READ` and `FD_WRITE`, as follows:

```
rc = WSAAsyncSelect(s, hWnd, wParam, 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, wParam1, FD_READ);  
rc = WSAAsyncSelect(s, hWnd, wParam2, FD_WRITE);
```

To cancel all notification indicating that Windows Sockets should send no further messages related to network events on the socket, *lEvent* is set to zero.

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

Although **WSAAsyncSelect** immediately disables event message posting for the socket in this instance, it is possible that messages could be waiting in the application's message queue. Therefore, the application must 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 still applies.

The socket created by the [accept](#) function has the same properties as the listening socket used to accept it. Consequently, **WSAAsyncSelect** events set for the listening socket also 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* parameter 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 that can be returned are as follows:

<b>Value</b>	<b>Meaning</b>
FD_READ	Socket <i>s</i> ready for reading
FD_WRITE	Socket <i>s</i> ready for writing
FD_OOB	Out-of-band data ready for reading on socket <i>s</i>
FD_ACCEPT	Socket <i>s</i> ready for accepting a new incoming connection
FD_CONNECT	Connection or multi-point <i>join</i> operation initiated on socket <i>s</i> completed
FD_CLOSE	Connection identified by socket <i>s</i> has been closed
FD_QOS	Quality of Service associated with socket <i>s</i> has changed
FD_GROUP_QOS	Quality of Service associated with the socket group to which <i>s</i> belongs has changed (reserved for future use with socket groups)
FD_ROUTING _INTERFACE_CHANGE	Local interface that should be used to send to the specified destination has changed
FD_ADDRESS_LIST _CHANGE	The list of addresses of the socket's protocol family to which the application client can bind has changed

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 that 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 **WS2\_32.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 that implicitly re-enables notification of that network event.

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

Any call to the re-enabling routine, even one that 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 can 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 quality of service change occurs. Further messages will *not* be forthcoming until either the provider detects a further change in quality of service or the application renegotiates the quality of service for the socket.

The **FD\_ROUTING\_INTERFACE\_CHANGE** message is posted when the local interface that should be used to reach the destination specified in **WSAIoctl** with **SIO\_ROUTING\_INTERFACE\_CHANGE** changes *after* such **IOCTL** has been issued.

The **FD\_ADDRESS\_LIST\_CHANGE** message is posted when the list of addresses to which the application can bind changes *after* **WSAIoctl** with **SIO\_ADDRESS\_LIST\_CHANGE** has been

issued.

If any event has already happened when the application calls **WSAAsyncSelect** or when the 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 [DECnet 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 can 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.

The **FD\_ROUTING\_INTERFACE\_CHANGE** and **FD\_ADDRESS\_LIST\_CHANGE** events are considered "edge triggered" as well. A message will be posted exactly once when a change occurs after the application has request the notification by issuing **WSAIoctl** with **SIO\_ROUTING\_INTERFACE\_CHANGE** or **SIO\_ADDRESS\_LIST\_CHANGE** correspondingly. Further messages will not be forthcoming until the application reissues the

IOCTL AND another change is detected since the IOCTL has been issued.

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

- **FD\_READ:**

1. when **WSAAsyncSelect** called, if there is data currently available to receive,
2. when data arrives, if **FD\_READ** not already posted,
3. after **recv** or **recvfrom** called (with or without **MSG\_PEEK**), if data is still available to receive.

**Note** when **setsockopt** **SO\_OOBINLINE** is enabled, "data" includes both normal data and out-of-band (OOB) data in the instances noted above.

- **FD\_WRITE:**

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

- **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),
3. after calling [WSAJoinLeaf](#), when join operation completes,
4. after [connect](#), [WSAConnect](#), or [WSAJoinLeaf](#) was called with a nonblocking, connection-oriented socket. The initial operation returned with a specific error of **WSAEWOULDBLOCK**, but the network operation went ahead. Whether the operation eventually succeeds or not, when the outcome has been determined, **FD\_CONNECT** happens. The client should check the error code to determine whether the outcome was successful or failed.

- **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 quality of service associated with the socket has been changed,
  2. after **WSAIoctl** with SIO\_GET\_QOS called, when the quality of service is changed.
- **FD\_GROUP\_QOS** (Reserved for future use with socket groups):
  1. when **WSAAsyncSelect** called, if the group quality of service associated with the socket has been changed,
  2. after **WSAIoctl** with SIO\_GET\_GROUP\_QOS called, when the group quality of service is changed.
- **FD\_ROUTING\_INTERFACE\_CHANGE:**
  1. after **WSAIoctl** with SIO\_ROUTING\_INTERFACE\_CHANGE called, when the local interface that should be used to reach the destination specified in the IOCTL changes.
- **FD\_ADDRESS\_LIST\_CHANGE**
  1. after **WSAIoctl** with SIO\_ADDRESS\_LIST\_CHANGE called, when the list of local addresses to which the application can bind changes.

## Return Values

If the **WSAAsyncSelect** function succeeds, the return value is zero provided 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](#).

## Error Codes

WSANOTINITIALISED	A successful <b>WSAStartup</b> 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 can 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

<b>Error Code</b>	<b>Meaning</b>
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 parameter 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**

<b>Error Code</b>	<b>Meaning</b>
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****Event: FD\_ADDRESS\_LIST\_CHANGE**

<b>Error Code</b>	<b>Meaning</b>
WSAENETDOWN	The network subsystem has failed.

**Event: FD\_ROUTING\_INTERFACE\_CHANGE**

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

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[select](#), [WSAEventSelect](#)

## WSACancelAsyncRequest

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

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

### Parameters

*hAsyncTaskHandle*

[in] The handle that specifies the asynchronous operation to be canceled.

### Remarks

The **WSACancelAsyncRequest** function is used to cancel an asynchronous operation that 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.

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.

### 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 can be retrieved by calling **WSAGetLastError**.

## Error Codes

WSANOTINITIALISED	A successful <b>WSAStartup</b> 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.

**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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[WSAAsyncGetHostByAddr](#), [WSAAsyncGetHostByName](#), [WSAAsyncGetProtoByName](#), [WSAAsyncGetProtoByNumber](#), [WSAAsyncGetServByName](#), [WSAAsyncGetServByPort](#)

# WSACancelBlockingCall

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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## WSACleanup

The Windows Sockets **WSACleanup** function terminates use of the WS2\_32.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 without 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.

Sockets that were open when **WSACleanup** was called are reset and automatically deallocated as if [closesocket](#) were called; sockets that have been closed with **closesocket** but that still have pending data to be sent can be affected—the pending data can be lost if the WS2\_32.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 unposted 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 WS2\_32.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 can be retrieved by calling **WSAGetLastError**.

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

In a multithreaded environment, **WSACleanup** terminates Windows Sockets operations for all threads.

### Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> 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.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[closesocket](#), [shutdown](#), [WSAStartup](#)

## WSACloseEvent

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

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

### Parameters

*hEvent*

[in] An object handle identifying the open event.

### 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 <a href="#">WSAStartup</a> 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	The <i>hEvent</i> is not a valid event object handle.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[WSACreateEvent](#), [WSAEnumNetworkEvents](#), [WSAEventSelect](#),  
[WSAGetOverlappedResult](#), [WSARecv](#), [WSARecvFrom](#), [WSAResetEvent](#), [WSASend](#),  
[WSASendTo](#), [WSASetEvent](#), [WSAWaitForMultipleEvents](#)

# WSAConnect

The Windows Sockets **WSAConnect** function establishes a connection to another socket application, exchanges connect data, and specifies needed quality of service based on the supplied **FLOWSPEC** structure.

```
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 socket in the other application to which to connect.

*namelen*

[in] The length of the *name*.

*lpCallerData*

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

*lpCalleeData*

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

*lpSQOS*

[in] A pointer to the **FLOWSPEC** structures for socket *s*, one for each direction.

*lpGQOS*

[in] Reserved for future use with socket groups. A pointer to the **FLOWSPEC** structures for the socket group (if applicable).

## Remarks

The **WSAConnect** 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. 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 connection-oriented, nonblocking sockets, it is often not possible to complete the connection immediately. In such cases, this function returns the error **WSAEWOULDBLOCK**. However, the operation proceeds. When the success or failure outcome becomes known, it may be reported in one of several ways depending on how the client registers for notification. If the client uses [select](#), success is reported in the *writelfds* set and failure is reported in the *exceptfds* set. If the client uses [WSAAsyncSelect](#) or [WSAEventSelect](#), the notification is announced with **FD\_CONNECT** and the error code associated with the **FD\_CONNECT** indicates either success or a specific reason for failure.

For a connectionless socket (for example, type **SOCK\_DGRAM**), the operation performed by **WSAConnect** is merely to establish a default destination address so that the socket can be used on subsequent connection-oriented send and receive operations ([send](#), [WSASend](#), [recv](#), and [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 disconnected. 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 **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* 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, **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 that 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 that will contain any user data passed back from the other socket as part of the connection establishment in a [WSABUF](#) structure. The member *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, the connection operation completes when the **WSAConnect** function returns. For nonblocking sockets, completion 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 can use the *lpSQOS* and *lpGQOS* parameters to override any previous quality of service specification made for the socket through [WSAIoctl](#) with either the SIO\_SET\_QOS or SIO\_SET\_GROUP\_QOS opcodes.

*lpSQOS* specifies the **FLOWSPEC** structures 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 quality of service 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 quality of service.

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

When connected sockets 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 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 can not be completed immediately. In this case, **WSAConnect** 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.
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 (successful or not).

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 (successful or not).

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 can call **WSAConnect** again for the same socket.

## Error Codes

<code>WSANOTINITIALISED</code>	A successful <a href="#">WSAStartup</a> must occur before using this function.
<code>WSAENETDOWN</code>	The network subsystem has failed.
<code>WSAEADDRINUSE</code>	The local address of the socket is already in use and the socket was not marked to allow address reuse with <code>SO_REUSEADDR</code> . This error usually occurs during the execution of <a href="#">bind</a> , but could be delayed until this function if the <b>bind</b> function operates on a partially wild-card address (involving <code>ADDR_ANY</code> ) and if a specific address needs to be "committed" at the time of this function.
<code>WSAEINTR</code>	The (blocking) Windows Socket 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .
<code>WSAEINPROGRESS</code>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<code>WSAEALREADY</code>	A nonblocking <a href="#">connect/WSAConnect</a> call is in progress on the specified socket.
<code>WSAEADDRNOTAVAIL</code>	The remote address is not a valid address (such as <code>ADDR_ANY</code> ).
<code>WSAEAFNOSUPPORT</code>	Addresses in the specified family cannot be used with this socket.
<code>WSAECONNREFUSED</code>	The attempt to connect was rejected.
<code>WSAEFAULT</code>	The <i>name</i> or the <i>namelen</i> parameter is not a valid part of the user address space, the <i>namelen</i> parameter is too small, the buffer length for <i>lpCalleeData</i> , <i>lpSQOS</i> , and <i>lpGQOS</i> are too small, or the buffer length for <i>lpCallerData</i> is too large.
<code>WSAEINVAL</code>	The parameter <i>s</i> is a listening socket, or the destination address specified is not consistent with that of the constrained group the socket belongs to.
<code>WSAEISCONN</code>	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 <b>FLOWSPEC</b> structures specified in <i>lpSQOS</i> and <i>lpGQOS</i> cannot be satisfied.
WSAEPROTONOSUPPORT	The <i>lpCallerData</i> argument 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.
WSAEACCES	Attempt to connect datagram socket to broadcast address failed because <b>setsockopt</b> SO_BROADCAST is not enabled.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[accept](#), [bind](#), [connect](#), [getsockname](#), [getsockopt](#), [select](#), [socket](#), [WSAAsyncSelect](#), [WSAEventSelect](#)

## WSACreateEvent

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

```
WSAEVENT WSACreateEvent(void);
```

### Remarks

The **WSACreateEvent** function is used to create an event object created that 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 can call the native [WSACreateEvent](#) Win32 function directly. The scope of an event object is limited to the process in which it is created.

### Return Values

If no error occurs, **WSACreateEvent** returns the handle of the event object. Otherwise, the return

value is `WSA_INVALID_EVENT`. To get extended error information, call [WSAGetLastError](#).

## Error Codes

<code>WSANOTINITIALISED</code>	A successful <a href="#">WSAStartup</a> must occur before using this function.
<code>WSAENETDOWN</code>	The network subsystem has failed.
<code>WSAEINPROGRESS</code>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<code>WSA_NOT_ENOUGH_MEMORY</code>	Not enough free memory available to create the event object.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in `winsock2.h`.

**Import Library:** Link with `ws2_32.lib`.

## See Also

[WSACloseEvent](#), [WSAEnumNetworkEvents](#), [WSAEventSelect](#), [WSAGetOverlappedResult](#), [WSARecv](#), [WSARecvFrom](#), [WSAResetEvent](#), [WSASend](#), [WSASendTo](#), [WSASetEvent](#), [WSAWaitForMultipleEvents](#)

# WSADuplicateSocket

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] A descriptor identifying the local socket.

*dwProcessId*

[in] The process ID of the target process in which the duplicated socket will be used.

*lpProtocolInfo*

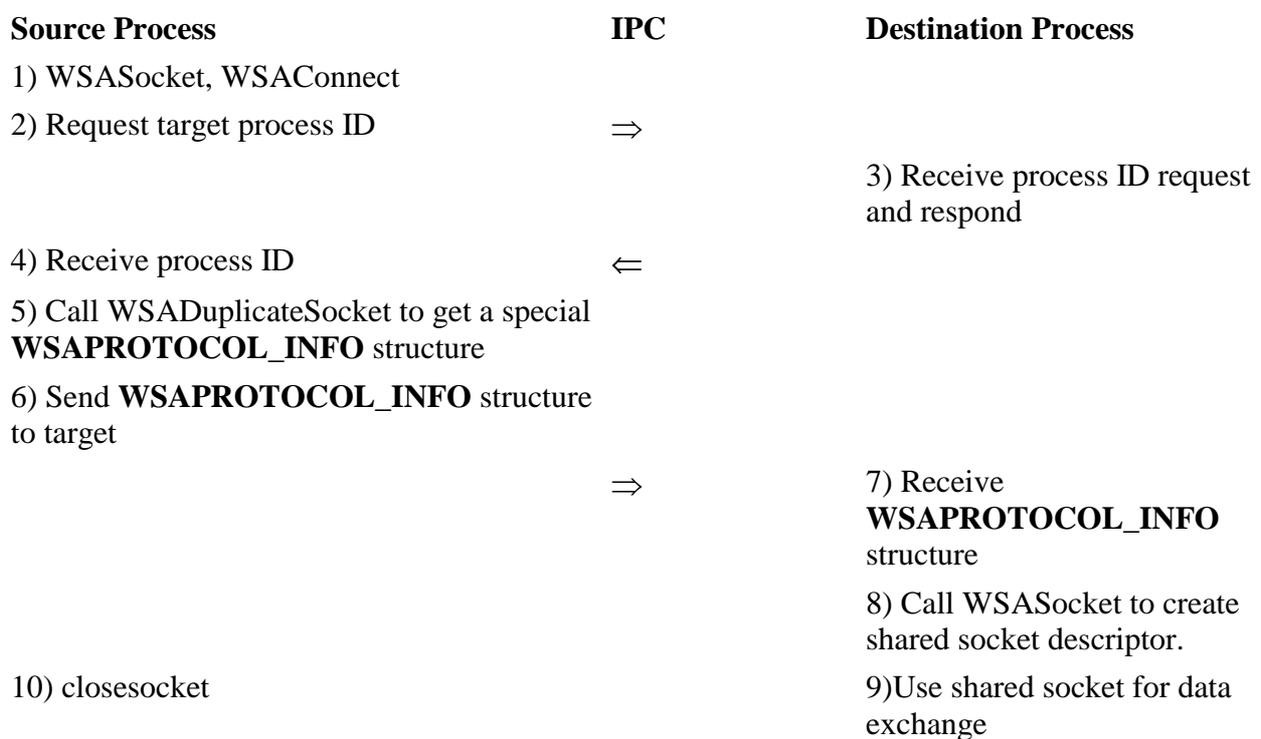
[out] A pointer to a buffer allocated by the client that is large enough to contain a [WSAPROTOCOL\\_INFO](#) structure. The service provider copies the protocol info structure contents to this buffer.

## Remarks

The **WSADuplicateSocket** 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. The special **WSAPROTOCOL\_INFO** structure can only be used once by the target process.

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

One possible scenario for establishing and handing off a shared socket is illustrated below:



The descriptors that reference a shared socket can be used independently for I/O. 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. Shared sockets are typically used to have one process that is responsible for creating sockets and establishing connections, and other processes that are responsible for information exchange.

All of the state information associated with a socket is held in common across all the descriptors because the socket descriptors are duplicated and not the actual socket. 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 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, developers would be advised to use threads instead of separate processes.

### Return Values

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

### Error Codes

<code>WSANOTINITIALISED</code>	A successful <a href="#">WSAStartup</a> must occur before using this function.
<code>WSAENETDOWN</code>	The network subsystem has failed.
<code>WSAEINVAL</code>	Indicates that one of the specified parameters was invalid.
<code>WSAEINPROGRESS</code>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<code>WSAEMFILE</code>	No more socket descriptors are available.
<code>WSAENOBUFS</code>	No buffer space is available. The socket cannot be created.
<code>WSAENOTSOCK</code>	The descriptor is not a socket.
<code>WSAEFAULT</code>	The <i>lpProtocolInfo</i> argument is not a valid part of the user address space.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in `winsock2.h`.

**Import Library:** Link with `ws2_32.lib`.

### See Also

[WSASocket](#)

## WSAEnumNameSpaceProviders

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

```
INT WINAPI WSAEnumNameSpaceProviders (
    IN OUT LPDWORD lpdwBufferLength,
    OUT 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 that is filled with [WSANAMESPACE\\_INFO](#) structures. 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

The **WSAEnumNameSpaceProviders** function returns the number of [WSANAMESPACE\\_INFO](#) structures copied into *lpnspBuffer*. Otherwise, the value SOCKET\_ERROR is returned, and a specific error number can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSAEFAULT	the buffer length was too small to receive all the relevant <a href="#">WSANAMESPACE_INFO</a> structures and associated information. Pass in a buffer at least as large as the value returned in <i>lpdwBufferLength</i> .
WSANOTINITIALIZED	The WS2_32.DLL has not been initialized. The application must first call <a href="#">WSAStartup</a> before calling any Windows Sockets functions.
WSA NOT ENOUGH MEMORY	There was insufficient memory to perform the operation.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

# WSAEnumNetworkEvents

The Windows Sockets **WSAEnumNetworkEvents** function discovers occurrences of network events for the indicated socket, clear internal network event records, and reset event objects (optional).

```
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 that is filled with a record of network events that occurred and any associated error codes.

## Remarks

The **WSAEnumNetworkEvents** 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. The 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](#).

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

The socket's internal record of network events is copied to the structure referenced by *lpNetworkEvents*, whereafter the internal network events record is cleared. If the *hEventObject* parameter is not 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 automatic, 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 *lNetworkEvents* member of the [WSANETWORKEVENTS](#) structure indicates which of the **FD\_XXX** network events have occurred. The *iErrorCode* array is used to contain any associated error codes with the array index corresponding to the position of event bits in *lNetworkEvents*. Identifiers such as **FD\_READ\_BIT** and **FD\_WRITE\_BIT** can be used to index the *iErrorCode* array. Note that only those elements of the *iErrorCode* array are set that correspond to the bits set in *lNetworkEvents* field. Other fields are not modified (this is important for backwards compatibility with the applications that are not aware of new **FD\_ROUTING\_INTERFACE\_CHANGE** and **FD\_ADDRESS\_LIST\_CHANGE** events).

The following error codes can be returned along with the corresponding network event:

**Event: FD\_CONNECT**

**Error Code**

WSAEAFNOSUPPORT

**Meaning**

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**

WSAENETDOWN

**Meaning**

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****Event: FD\_ADDRESS\_LIST\_CHANGE****Error Code**

WSAENETDOWN

**Meaning**

The network subsystem has failed.

**Event: FD\_ROUTING\_INTERFACE\_CHANGE****Error Code**

WSAENETUNREACH

**Meaning**

The specified destination is no longer reachable

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 can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSANOTINITIALISED	A successful <b>WSAStartup</b> 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.
WSAEFAULT	The <i>lpNetworkEvents</i> argument is not a valid part of the user address space.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

### [WSAEventSelect](#)

# WSAEnumProtocols

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 that is filled with [WSAPROTOCOL\\_INFO](#) structures.

### *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

The **WSAEnumProtocols** 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. Often, *lpiProtocols* will be supplied as a NULL pointer that will cause the function 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 **WSAEnumProtocols** 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 using the WS2\_32.DLL, or with any subsequent re-ordering that can have occurred through the Windows Sockets applet or DLL supplied for establishing default TCP/IP providers.

## Return Values

If no error occurs, **WSAEnumProtocols** returns the number of protocols to be reported. Otherwise, a value of SOCKET\_ERROR is returned and a specific error code can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> 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 <b>WSAPROTOCOL_INFO</b> structures and associated information. Pass in a buffer at least as large as the value returned in <i>lpdwBufferLength</i> .
WSAEFAULT	One or more of the <i>lpiProtocols</i> , <i>lpProtocolBuffer</i> , or <i>lpdwBufferLength</i> arguments are not a valid part of the user address space.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

# WSAEventSelect

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 that specifies the combination of FD\_XXX network events in which the application has interest.

## Remarks

The **WSAEventSelect** 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 the *s* parameter. The event object is set when any of the nominated network events occur.

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

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

The **WSAEventSelect** 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 or multipoint "join" operation
FD_CLOSE	Want to receive notification of socket closure
FD_QOS	Want to receive notification of socket Quality of Service (QOS) changes
FD_GROUP_QOS	Reserved for future use with socket groups. Want to receive notification of socket group Quality of Service (QOS) changes
FD_ROUTING _INTERFACE_CHANGE	Want to receive notification of routing interface changes for the specified destination
FD_ADDRESS_LIST _CHANGE	Want to receive notification of local address list changes for the address family of the socket

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.

The socket created when the **accept** function is called 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 *hEventObject* 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 can 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 that returns WSAEWOLDBLOCK immediately. For example, the following sequence of operations is possible:

1. Data arrives on socket *s*; Windows Sockets sets the **WSAEventSelect** event object.
2. The 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. The application issues a **recv**(*s*,...) to read the data.
5. The application eventually waits on event object specified in **WSAEventSelect**, which returns immediately indicating that data is ready to read.
6. The application issues **recv**(*s*,...), which fails with the error WSAEWOLDBLOCK.

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 that implicitly re-enables the setting of that network event and signaling of the associated event object.

Network Event	Re-enabling function
FD_READ	<a href="#">recv</a> , <a href="#">recvfrom</a> , <a href="#">WSARecv</a> , or <a href="#">WSARecvFrom</a>
FD_WRITE	<a href="#">send</a> , <a href="#">sendto</a> , <a href="#">WSASend</a> , or <a href="#">WSASendTo</a>
FD_OOB	<b>recv</b> , <b>recvfrom</b> , <b>WSARecv</b> , or <b>WSARecvFrom</b>
FD_ACCEPT	<a href="#">accept</a> or <a href="#">WSAAccept</a> unless the error code returned is WSATRY_AGAIN indicating that the condition function returned CF_DEFER
FD_CONNECT	NONE
FD_CLOSE	NONE
FD_QOS	<a href="#">WSAIoctl</a> with command SIO_GET_QOS

FD_GROUP_QOS	Reserved for future use with socket groups. <b>WSAIoctl</b> with command SIO_GET_GROUP_QOS
FD_ROUTING_INTERFACE_CHANGE	<b>WSAIoctl</b> with command SIO_ROUTING_INTERFACE_CHANGE
FD_ADDRESS_LIST_CHANGE	<b>WSAIoctl</b> with command SIO_ADDRESS_LIST_CHANGE

Any call to the re-enabling routine, even one that fails, results in re-enabling of recording and signaling for the relevant network event and event object.

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 WS2\_32.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 WS2\_32.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 quality of service change occurs. Further messages will *not* be forthcoming until either the provider detects a further change in quality of service or the application renegotiates the quality of service for the socket.

The FD\_ROUTING\_INTERFACE\_CHANGE and FD\_ADDRESS\_LIST\_CHANGE events are considered "edge triggered" as well. A message will be posted exactly once when a change occurs AFTER the application has request the notification by issuing **WSAIoctl** with SIO\_ROUTING\_INTERFACE\_CHANGE or SIO\_ADDRESS\_LIST\_CHANGE correspondingly. Further messages will not be forthcoming until the application reissues the IOCTL AND another change is detected since the IOCTL has been issued.

If a network event has already happened when the application calls **WSAEventSelect** or when the 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 can 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 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 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.

The FD\_ROUTING\_INTERFACE\_CHANGE network event is recorded when the local interface that should be used to reach the destination specified in **WSAIoctl** with SIO\_ROUTING\_INTERFACE\_CHANGE changes *after* such IOCTL has been issued.

The FD\_ADDRESS\_LIST\_CHANGE network event is recorded when the list of addresses of protocol family for the socket to which the application can bind changes *after* **WSAIoctl** with SIO\_ADDRESS\_LIST\_CHANGE has been issued.

## Error Codes

WSANOTINITIALISED	A successful <b>WSAStartup</b> 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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[WSAAsyncSelect](#), [WSACloseEvent](#), [WSACreateEvent](#), [WSAEnumNetworkEvents](#), [WSAWaitForMultipleEvents](#)

# WSAGetLastError

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

```
int WSAGetLastError (void);
```

## Remarks

The **WSAGetLastError** 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 can be different from the error code obtained from [getsockopt](#) SO\_ERROR, which is socket-specific since **WSAGetLastError** is for all thread-specific sockets.

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. A [getsockopt](#) SO\_ERROR also resets the error code to zero.

The **WSAGetLastError** 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 can 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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[getsockopt](#), [WSASetLastError](#)

# WSAGetOverlappedResult

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] A descriptor identifying the socket. This is the same socket that was specified when the overlapped operation was started by a call to [WSARecv](#), [WSARecvFrom](#), [WSASend](#), [WSASendTo](#), or [WSAIoctl](#).

*lpOverlapped*

[in] A pointer to a [WSAOVERLAPPED](#) structure that was specified when the overlapped operation was started.

*pcbTransfer*

[out] A pointer 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] A flag that specifies whether the function should wait for the pending overlapped operation to complete. If TRUE, the function does not return until the operation has been completed. If FALSE and the operation is still pending, the function returns FALSE and the [WSAGetLastError](#) function returns WSA\_IO\_INCOMPLETE. The *fWait* parameter may be set to TRUE only if the overlapped operation selected the event-based completion notification.

*lpdwFlags*

[out] A pointer 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 **WSAGetOverlappedResult** function reports results of last overlapped operation for the specified socket. The **WSAOverlappedResult** function is passed the socket descriptor and the [WSAOVERLAPPED](#) structure that were specified when the overlapped function was called. 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 such as [WSARecv](#) 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 completed, the system sets the event object to the signaled state.

If the *fWait* parameter is TRUE, **WSAGetOverlappedResult** determines whether the pending operation has been completed by waiting for the event object to be in the signaled state. A client may set *fWait* parameter to TRUE only if it selected event-based completion notification when the IO operation was requested. If another form of notification was selected, the usage of the *hEvent* parameter of the **WSAOVERLAPPED** structure is different, and setting *fWait* to TRUE causes unpredictable results.

### 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, the overlapped operation completed but with errors, or the overlapped operation's 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 <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSA_INVALID_HANDLE	The <i>hEvent</i> field of the <a href="#">WSAOVERLAPPED</a> structure does not contain a valid event object handle.
WSA_INVALID_PARAMETER	One of the parameters is unacceptable.
WSA_IO_INCOMPLETE	The <i>fWait</i> parameter is FALSE and the I/O operation has not yet completed.
WSAEFAULT	One or more of the <i>lpOverlapped</i> , <i>lpcbTransfer</i> , or <i>lpdwFlags</i> arguments are not a valid part of the user address space.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[WSAAccept](#), [WSAConnect](#), [WSACreateEvent](#), [WSAIoctl](#), [WSARecv](#), [WSARecvFrom](#), [WSASend](#), [WSASendTo](#), [WSAWaitForMultipleEvents](#)

# WSAGetQOSByName

The Windows Sockets **WSAGetQOSByName** function initializes a **QUALITYOFSERVICE** structure based on a named template, or it supplies a buffer to retrieve an enumeration of the available template names.

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

## Parameters

*s*  
[in] A descriptor identifying a socket.

*lpQOSName*  
[in out] A pointer to a specific quality of service template.

*lpQOS*  
[out] A pointer to the [QUALITYOFSERVICE](#) structure to be filled.

## Remarks

The **WSAGetQOSByName** function is used by applications to initialize a [QUALITYOFSERVICE](#) structure to a set of known values appropriate for a particular service class or media type. These values are stored in a template that is referenced by a well-known name. The client may retrieve these values by setting the *buf* parameter of the WSABUF indicated by *lpQOSName*, which points to a string of non-zero length specifying a template name. In this case, the usage of *lpQOSName* is IN only, and results are returned through *lpQOS*.

Alternatively, the client may use this function to retrieve an enumeration of available template names. The client may do this by setting the *buf* parameter of the WSABUF indicated by *lpQOSName* to a zero-length null-terminated string. In this case the buffer indicated by *buf* is over-written with a sequence of as many null-terminated template names that are available up to the number of bytes available in *buf* as indicated by the *len* parameter of the WSABUF indicated by *lpQOSName*. The list of names itself is terminated by a zero-length name. When the [WSAGetQOSByName](#) function is used to retrieve template names, the *lpQOS* parameter is ignored.

## Return Values

If **WSAGetQOSByName** 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 <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lpQOSName</i> or <i>lpQOS</i> parameter are not a valid part of the user address space, or the buffer length for <i>lpQOS</i> is too small.
WSAENVAL	The specified quality of service template name is invalid.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[getsockopt](#), [WSAAccept](#), [WSAConnect](#)

# WSAGetServiceClassInfo

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 lpdwBufferLength,
    LPWSASERVICECLASSINFO lpServiceClassInfo
);

```

## Parameters

*lpProviderId*

[in] A pointer to a GUID that identifies a specific name space provider

*lpServiceClassId*

[in] A 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] A pointer to the service class information from the indicated name space provider for the specified service class.

## Remarks

The **WSAGetServiceClassInfo** function retrieves service class information but the service class information retrieved from a particular name space provider might not 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 **WSAGetServiceClassInfo** was successful. Otherwise, the value `SOCKET_ERROR` is returned, and a specific error number can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSAEACCESS	The calling routine does not have sufficient privileges to access the information.
WSAEFAULT	The buffer referenced by <i>lpServiceClassInfo</i> is too small. Pass in a larger buffer.
WSAEINVAL	the specified service class ID or name space provider ID is invalid.
WSANOTINITIALIZED	The WS2_32.DLL has not been initialized. The application must first call <a href="#">WSAStartup</a> before calling any Windows Sockets functions.
WSATYPE NOT FOUND	The specified class was not found.
WSA NOT ENOUGH MEMORY	There was insufficient memory to perform the operation.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

# **WSAGetServiceClassNameByClassId** The Windows Sockets **WSAGetServiceClassNameByClassId** 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 WSAGetServiceClassNameByClassId(  
    LPGUID lpServiceClassId,  
    LPTSTR lpszServiceClassName,  
    LPDWORD lpdwBufferLength  
);
```

## Parameters

*lpServiceClassId*

[in] A pointer to the GUID for the service class.

*lpszServiceClassName*

[out] A pointer to the service name.

*lpdwBufferLength*

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

## Return Values

The **WSAGetServiceClassNameByClassId** function returns a value of zero if successful. Otherwise, the value `SOCKET_ERROR` is returned, and a specific error number can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSAEFAULT	The specified buffer referenced by <i>lpszServiceClassName</i> is too small. Pass in a larger buffer.
WSA_INVALID_PARAMETER	The <i>lpServiceClassId</i> parameter specified is invalid.
WSANOTINITIALIZED	The WS2_32.DLL has not been initialized. The application must first call <a href="#">WSAStartup</a> before calling any Windows Sockets functions.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

# WSAHtonl

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

The **WSAHtonl** function 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 can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lpnetlong</i> parameter is not completely contained in a valid part of the user address space.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in `winsock2.h`.

**Import Library:** Link with `ws2_32.lib`.

## See Also

[htonl](#), [htons](#), [ntohl](#), [ntohs](#), [WSANTohl](#), [WSAHtons](#), [WSANTohs](#)

## WSAHtons

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

The **WSAHtons** function 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 can be retrieved by calling [WSAGetLastError](#).

### Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lpnetshort</i> parameter is not completely contained in a valid part of the user address space.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[htonl](#), [htons](#), [ntohl](#), [ntohs](#), [WSAhtonl](#), [WSANtohl](#), [WSANtohs](#)

# WSAInstallServiceClass

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] The 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 can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSAEACCES	The calling function 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 <a href="#">WSARemoveServiceClass</a> , and then re-install with updated class info data.
WSAEINVAL	The service class information was invalid or improperly structured.
WSANOTINITIALIZED	The WS2_32.DLL has not been initialized. The application must first call <a href="#">WSAStartup</a> before calling any Windows Sockets functions.
WSA NOT ENOUGH MEMORY	There was insufficient memory to perform the operation.

## QuickInfo

**Windows NT:** Yes  
**Windows:** Yes  
**Windows CE:** Unsupported.  
**Header:** Declared in winsock2.h.  
**Import Library:** Link with ws2\_32.lib.

## WSAIoctl

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] A descriptor identifying a socket.

*dwIoControlCode*  
 [in] The control code of operation to perform.

*lpvInBuffer*  
 [in] A pointer to the input buffer.

*cbInBuffer*  
 [in] A number indicating the size of the input buffer.

*lpvOutBuffer*  
 [out] A pointer to the output buffer.

*cbOutBuffer*  
 [in] A number indicating the size of the output buffer.

*lpcbBytesReturned*  
 [out] A pointer to actual number of bytes of output.

*lpOverlapped*  
 [in] A pointer to a [WSAOVERLAPPED](#) structure (ignored for non-overlapped sockets).

*lpCompletionRoutine*  
 [in] A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

### Remarks

The **WSAIoctl** function is used to set or retrieve operating parameters associated with the socket, the transport protocol, or the communications subsystem.

If both *lpOverlapped* and *lpCompletionRoutine* are NULL, the socket in this function will be

treated as a non-overlapped socket. For a non-overlapped socket, *lpOverlapped* and *lpCompletionRoutine* parameters are ignored, which cause the function to behave like the standard [ioctlsocket](#) function except that **WSAIoctl** can block if socket *s* is in the blocking mode. If socket *s* is in the nonblocking mode, this function can return **WSAEWOULDBLOCK** when the specified operation cannot be finished immediately. In this case, the application may change the socket to the blocking mode and reissue the request or wait for the corresponding network event (such as **FD\_ROUTING\_INTERFACE\_CHANGE** or **FD\_ADDRESS\_LIST\_CHANGE** in case of **SIO\_ROUTING\_INTERFACE\_CHANGE** or **SIO\_ADDRESS\_LIST\_CHANGE**) using Windows message (using **WSAAsyncSelect**) or event (using **WSAEventSelect**) based notification mechanism

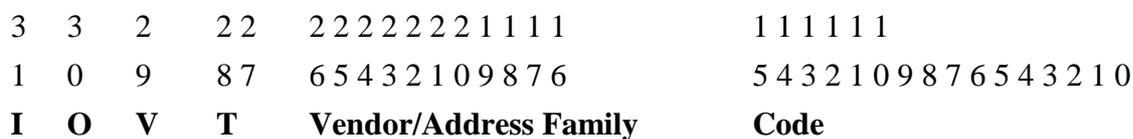
For overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The final completion status is retrieved through [WSAGetOverlappedResult](#). The *lpcbBytesReturned* parameter is ignored.

Any *ioctl* may block indefinitely, depending on the service provider's implementation. If the application cannot tolerate blocking in a **WSAIoctl** call, overlapped I/O would be advised for *ioctls* that are especially likely to block including:

- SIO\_FINDROUTE
- SIO\_FLUSH
- SIO\_GET\_QOS
- SIO\_GET\_GROUP\_QOS
- SIO\_SET\_QOS
- SIO\_SET\_GROUP\_QOS
- SIO\_ROUTING\_INTERFACE\_CHANGE
- SIO\_ADDRESS\_LIST\_CHANGE

Some protocol-specific *ioctls* may also be especially likely to block. Check the relevant protocol-specific annex for any available information.

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 in as much as the *dwIoControlCode* parameter is now a 32-bit entity. 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:



**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**. Codes with both input and output parameters set both **I** and **O**.

**V** is set if there are no parameters for the code, as with **IOC\_VOID**.

**T** is a two-bit quantity that 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 that appears in the **Vendor/Address family** field. 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 that defines the vendor who owns the code (if `T == 3`) or that 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 16-bit quantity that contains the specific `ioctl` code for the operation.

The following Unix `ioctl` codes (commands) are supported:

#### 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 that 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 that can be read in a single receive operation; this is normally the same as the total amount of data queued on the socket (since data stream is byte-oriented, this is not guaranteed). 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`) that has been configured for in-line reception of any out-of-band data (`SO_OOINLINE`). 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:

**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 that 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 (a separate document) 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 that do not support this ioctl. The handle associated by this ioctl can be retrieved using **SIO\_TRANSLATE\_HANDLE**.

A companion interface might be used, for example, if a particular provider provides (1) a great deal of additional controls over the behavior of a socket and (2) the controls are provider-specific enough that they do not map to existing Windows Socket functions or ones likely to be defined in the future. It is recommended that the Component Object Model (COM) be used instead of this ioctl to discover and track other interfaces that might be supported by a socket. This ioctl is present for (reverse) compatibility with systems where COM is not available or cannot be used for some other reason.

**SIO\_ENABLE\_CIRCULAR\_QUEUEING** (opcode setting: V, T==1)

Indicates to the underlying message-oriented service provider that a newly arrived message should never be dropped because of a buffer queue overflow. Instead, the oldest message in the queue should be eliminated in order to accommodate the newly arrived message. No input and output buffers are required. Note that this ioctl is only valid for sockets associated with unreliable, message-oriented protocols. The **WSAENOPROTOOPT** error code is indicated for service providers that 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 that 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)

Reserved for future use with sockets. Retrieve the **QUALITYOFSERVICE** 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 quality of service request. The

**QUALITYOFSERVICE** structure will be copied into the output buffer. The output buffer must be sized large enough to be able to contain the full **QUALITYOFSERVICE** structure. The WSAENOPROTOOPT error code is indicated for service providers that do not support quality of service.

**SIO\_GET\_GROUP\_QOS** (opcode setting: O, I, T==1)

Retrieve the **QUALITYOFSERVICE** 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 quality of service request. The **QUALITYOFSERVICE** 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 **QUALITYOFSERVICE** structure are set to NULL. The WSAENOPROTOOPT error code is indicated for service providers that do not support quality of service.

**SIO\_MULTIPPOINT\_LOOPBACK** (opcode setting: I, T==1)

Controls whether data sent in a multipoint session will also be received by the same socket on the local host. A value of TRUE causes loopback reception to occur while a value of FALSE prohibits this. By default, loopback is enabled.

**SIO\_MULTICAST\_SCOPE** (opcode setting: I, T==1)

Specifies the scope over which multicast transmissions will occur. Scope is defined as the number of routed network segments to be covered. A scope of zero would indicate that the multicast transmission would not be placed "on the wire" but could be disseminated across sockets within the local host. A scope value of one (the default) indicates that the transmission will be placed on the wire, but will *not* cross any routers. Higher scope values determine the number of routers that can be crossed. Note that this corresponds to the time-to-live (TTL) parameter in IP multicasting. By default, scope is 1.

**SIO\_SET\_QOS** (opcode setting: I, T==1)

Associate the supplied **QUALITYOFSERVICE** structure with the socket. No output buffer is required, the **QUALITYOFSERVICE** structure will be obtained from the input buffer. The WSAENOPROTOOPT error code is indicated for service providers that do not support quality of service.

**SIO\_SET\_GROUP\_QOS**(opcode setting: I, T==1)

Reserved for future use with sockets. Establish the supplied **QUALITYOFSERVICE** structure with the socket group to which this socket belongs. No output buffer is required, the **QUALITYOFSERVICE** structure will be obtained from the input buffer. The WSAENOPROTOOPT error code is indicated for service providers that do not support quality of service, 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 Windows Sockets 2 Protocol-Specific Annex for details specific to a particular companion interface. The WSAENOPROTOOPT error code is indicated for service providers that do not support this ioctl for the specified companion interface. This ioctl retrieves the handle associated using **SIO\_TRANSLATE\_HANDLE**.

It is recommend that the Component Object Model (COM) be used instead of this ioctl to discover and track other interfaces that might be supported by a socket. This ioctl is present for (reverse) compatibility with systems where COM is not available or cannot be used for some other reason.

**SIO\_ROUTING\_INTERFACE\_QUERY** (opcode setting: I, O, T==1)

To obtain the address of the local interface (represented as SOCKADDR structure) which should be used to send to the remote address specified in the input buffer (as SOCKADDR). Remote multicast addresses may be submitted in the input buffer to get the address of the preferred interface for multicast transmission. In any case, the interface address returned may be used by the application in a subsequent bind() request.

Note that routes are subject to change. Therefore, applications cannot rely on the information returned by SIO\_ROUTING\_INTERFACE\_QUERY to be persistent. Applications may register for routing change notifications via the SIO\_ROUTING\_INTERFACE\_CHANGE IOCTL which provides for notification via either overlapped IO or FD\_ROUTING\_INTERFACE\_CHANGE event. The following sequence of actions can be used to guarantee that the application always has current routing interface information for a given destination:

- issue SIO\_ROUTING\_INTERFACE\_CHANGE IOCTL
- issue SIO\_ROUTING\_INTERFACE\_QUERY IOCTL
- whenever SIO\_ROUTING\_INTERFACE\_CHANGE IOCTL notifies the application of routing change (either via overlapped IO or by signaling FD\_ROUTING\_INTERFACE\_CHANGE event), the whole sequence of actions should be repeated.

If output buffer is not large enough to contain the interface address, SOCKET\_ERROR is returned as the result of this IOCTL and **WSAGetLastError** returns WSAEFAULT. The required size of the output buffer will be returned in *lpcbBytesReturned* in this case. Note the WSAEFAULT error code is also returned if the *lpvInBuffer*, *lpvOutBuffer* or *lpcbBytesReturned* parameter is not totally contained in a valid part of the user address space.

If the destination address specified in the input buffer cannot be reached via any of the available interfaces, SOCKET\_ERROR is returned as the result of this IOCTL and **WSAGetLastError** returns WSAENETUNREACH or even WSAENETDOWN if all of the network connectivity is lost.

SIO\_ROUTING\_INTERFACE\_CHANGE (opcode setting: I, T==1)

To receive notification of the interface change that should be used to reach the remote address in the input buffer (specified as a SOCKADDR structure). No output information will be provided upon completion of this IOCTL; the completion merely indicates that routing interface for a given destination has changed and should be queried again via SIO\_ROUTING\_INTERFACE\_QUERY.

It is assumed (although not required) that the application uses overlapped IO to be notified of routing interface change via completion of SIO\_ROUTING\_INTERFACE\_CHANGE request. Alternatively, if the SIO\_ROUTING\_INTERFACE\_CHANGE IOCTL is issued on non-blocking socket *and* without overlapped parameters (*lpOverlapped* / *CompletionRoutine* are set NULL), it will complete immediately with error WSAEWOULD\_BLOCK, and the application can then wait for routing change events via call to **WSAEventSelect** or **WSAAsyncSelect** with FD\_ROUTING\_INTERFACE\_CHANGE bit set in the network event bitmask

It is recognized that routing information remains stable in most cases so that requiring the application to keep multiple outstanding IOCTLs to get notifications about all destinations that it is interested in as well as having service provider to keep track of all them will unnecessarily tie significant system resources. This situation can be avoided by extending the meaning of the input parameters and relaxing the service provider requirements as follows:

- The application can specify a protocol family specific wildcard address (same as one used in **bind** call when requesting to bind to any available address) to request notifications of any routing changes. This allows the application to keep only one outstanding `SIO_ROUTING_INTERFACE_CHANGE` for all the sockets/destinations it has and then use `SIO_ROUTING_INTERFACE_QUERY` to get the actual routing information
- Service provider has the option to ignore the information supplied by the application in the input buffer of the `SIO_ROUTING_INTERFACE_CHANGE` (as though the application specified a wildcard address) and complete the `SIO_ROUTING_INTERFACE_CHANGE` IOCTL or signal `FD_ROUTING_INTERFACE_CHANGE` event in the event of any routing information change (not just the route to the destination specified in the input buffer).

`SIO_ADDRESS_LIST_QUERY` (opcode setting: I, O, T==1)

To obtain a list of local transport addresses of socket's protocol family to which the application can bind. The list returned in the output buffer using the following format:

```
typedef struct _SOCKET_ADDRESS_LIST {
    INT iAddressCount;
    SOCKET_ADDRESS Address[1];
} SOCKET_ADDRESS_LIST, FAR * LPSOCKET_ADDRESS_LIST;
Members:
    iAddressCount - number of address structures in the list;
    Address - array of protocol family specific address structures.
```

Note that in Win32 Plug-n-Play environments addresses can be added/removed dynamically. Therefore, applications cannot rely on the information returned by `SIO_ADDRESS_LIST_QUERY` to be persistent. Applications may register for address change notifications via the `SIO_ADDRESS_LIST_CHANGE` IOCTL which provides for notification via either overlapped IO or `FD_ADDRESS_LIST_CHANGE` event. The following sequence of actions can be used to guarantee that the application always has current address list information:

- issue `SIO_ADDRESS_LIST_CHANGE` IOCTL

- issue `SIO_ADDRESS_LIST_QUERY` IOCTL
- whenever `SIO_ADDRESS_LIST_CHANGE` IOCTL notifies the application of address list change (either via overlapped IO or by signaling `FD_ADDRESS_LIST_CHANGE` event), the whole sequence of actions should be repeated.

If output buffer is not large enough to contain the address list, `SOCKET_ERROR` is returned as the result of this IOCTL and `WSAGetLastError` returns `WSAEFAULT`. The required size of the output buffer will be returned in `lpcbBytesReturned` in this case. Note the `WSAEFAULT` error code is also returned if the `lpvInBuffer`, `lpvOutBuffer` or `lpcbBytesReturned` parameter is not totally contained in a valid part of the user address space.

`SIO_ADDRESS_LIST_CHANGE` (opcode setting: T==1)

To receive notification of changes in the list of local transport addresses of socket's protocol family to which the application can bind. No output information will be provided upon completion of this IOCTL; the completion merely indicates that list of available local address has changed and should be queried again via `SIO_ADDRESS_LIST_QUERY`.

It is assumed (although not required) that the application uses overlapped IO to be notified of change via completion of `SIO_ADDRESS_LIST_CHANGE` request. Alternatively, if the `SIO_ADDRESS_LIST_CHANGE` IOCTL is issued on non-blocking socket AND without overlapped parameters (`lpOverlapped` / `lpCompletionRoutine` are set to NULL), it will complete immediately with error `WSAEWOULDBLOCK`. The application can then wait

for address list change events via call to **WSAEventSelect** or **WSAAsyncSelect** with **FD\_ADDRESS\_LIST\_CHANGE** bit set in the network event bitmask.

If an overlapped operation completes immediately, **WSAIoctl** 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 *lpOverlapped* parameter contains the address of a [WSAOVERLAPPED](#) structure.

If the *lpCompletionRoutine* parameter is **NULL**, the *hEvent* field of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use [WSAWaitForMultipleEvents](#) or [WSAGetOverlappedResult](#) to wait or poll on the event object.

If *lpCompletionRoutine* is not **NULL**, the *hEvent* field is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-**NULL** *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped IO request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to **TRUE**. In this case the usage of the *hEvent* field is undefined, and attempting to wait on the *hEvent* field would produce unpredictable results.

The prototype of the completion routine is as follows:

```
void CALLBACK CompletionRoutine(  
    IN  DWORD  dwError,  
    IN  DWORD  cbTransferred,  
    IN  LPWSAOVERLAPPED lpOverlapped,  
    IN  DWORD  dwFlags  
);
```

This **CompletionRoutine** is a placeholder for an application-defined or library-defined function. The *dwError* parameter specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* parameter specifies the number of bytes returned. Currently, there are no flag values defined and *dwFlags* will be zero. The **CompletionRoutine** function does not return a value.

Returning from this function allows invocation of another pending completion routine for this socket. The completion routines can be called in any order, not necessarily in the same order the overlapped operations are completed.

**Windows CE:** For secure sockets, the *lpcbBytesReturned* parameter is a pointer to a **DWORD** receiving the number of bytes returned in output buffer. Also, the last two parameters, namely, *lpOverlapped* and *lpCompletionRoutine*, must be **NULL**.

In addition to the **ioctlsocket** control codes (**FIONBIO**, **SIOCATMARK**, **FIONREAD**), the *dwIoControlCode* parameter may also assume the following SSL-specific control flags:

<b>Value</b>	<b>Meaning</b>
SO_SSL_GET_CAPABILITIES	Retrieves a set of flags describing the WinSock security capabilities. The output buffer must be a pointer to a DWORD field. At present, only the SO_CAP_CLIENT flag is defined.
SO_SSL_GET_FLAGS	Retrieves s-channel specific flags associated with a particular socket. The output buffer must be a pointer to a DWORD field. See SO_SSL_SET_FLAGS for details on valid flags.
SO_SSL_SET_FLAGS	Sets the sockets' current s-channel-specific flag values. The output buffer must be a pointer to a DWORD bit field. Currently, the SO_SSL_FLAG_DEFER_HANDSHAKE flag is defined to allow an application to send and receive plain text data before sending cipher text. It is required for setting up communication with proxy servers.
SO_SSL_PERFORM_HANDSHAKE	Normally the WinSock security provider performs the handshake in the WinSock connect API. However, if this flag is set, the handshake is deferred until the application issues the SO_SSL_PERFORM_HANDSHAKE control code. After successful handshaking, this flag is reset.
SO_SSL_GET_PROTOCOLS	Retrieves a list of protocols that the provider currently supports on this socket. The output buffer must be a pointer to an SSLPROTOCOLS structure as described below: <pre> typedef struct _SSLPROTOCOL {     DWORD dwProtocol;     DWORD dwVersion;     DWORD dwFlags; } SSLPROTOCOL, *LPSSLPROTOCOL; typedef struct _SSLPROTOCOLS {     DWORD dwCount;     SSLPROTOCOL ProtocolList[1]; } SSLPROTOCOLS, FAR *LPSSLPROTOCOLS; </pre>
SO_SSL_SET_PROTOCOLS	Valid protocols include SSL_PROTOCOL_SSL2, SSL_PROTOCOL_SSL3, and SSL_PROTOCOL_PC. Specifies a list of protocols the provider is to support on this socket. The input buffer must be a pointer to an SSLPROTOCOLS structure described above.
SO_SSL_SET_VALIDATE_CERT_HOOK	Sets the pointer to the socket's certificate validation hook. This hook is used to specify the callback function invoked by the WinSock security provider when a set of credentials is received from the other party. The input buffer must be a pointer to the SSLVALIDATECERTHOOK structure, described below: <pre> typedef struct {     SSLVALIDATECERTFUNC HookFunc;     LPVOID pvArg; } SSLVALIDATEPROTOCOLCERTHOOK, *PSSLVALIDATECERTHOOK; </pre>
	where <i>HookFunc</i> is a pointer to the certificate validation callback function (see "Certificate Validation Callback" below), <i>pvArg</i> is a pointer to application specific data and may be used by the application for any purpose.

**SO\_SSL\_PERFORM\_HANDSHAKE** Initiates the secure handshake sequence on a connecte where the **SSL\_FLAG\_DEFER\_HANDSHAKE** flag l prior to the connection. Data buffers are not required, **SSL\_FLAG\_DEFER\_HANDSHAKE** flag will be rese

### Certificate Validation Callback

The WinSock security provider invokes the certificate validation callback when the remote party receives a certificate for server authentication. All the client applications must implement the callback function to ensure that the certificate should meet the following minimum requirements:

- The certificate has not expired;
- The identity contained within the certificate matches that of the remote party.

The certificate validation callback function is of the following type:

```
typedef int (CALLBACK *SSLVALIDATECERTFUNC){
    DWORD    dwType,        // in
    LPVOID   pvArg,        // in
    DWORD    dwChainLen,   // in
    LPBLOB   pCertChain,   // in
    DWORD    dwFlags       // in
};
```

The parameters are defined in the following table.

Parameters	Meaning
<i>dwType</i>	Specifies the type of data pointed to be <i>pCertChain</i> . This must be <b>SSL_CERT_X59</b> to specify that <i>pCertChain</i> be a pointer to an X509 style certificate.
<i>PvArg</i>	An application-defined context passed into the <b>SSLVALIDATECERTHOOK</b> structure.
<i>DwChainLen</i>	The number of certificates pointed to by <i>pCertChain</i> . In Windows CE this will always be one.
<i>PCertChain</i>	Pointer to the remote party's certificate.
<i>DwFlags</i>	To be designed to indicate that the certificate issuer list has been checked with the list of known certificate authorities and that the certificate is either trusted or not. This is required since not all the certificate chain is passed to the application.

The application-defined callback function will typically return one of the following error codes:

Value	Meaning
<b>SSL_ERR_OKAY</b>	The remote party's certificate is acceptable.
<b>SSL_ERR_BAD_DATA</b>	The certificate is improperly formatted.
<b>SSL_ERR_BAD_SIG</b>	The signature check fails.
<b>SSL_ERR_CERT_EXPIRED</b>	The certificate has expired.
<b>SSL_ERR_CERT_REVOKED</b>	The certificate has been revoked by its issuer.

**SSL\_ERR\_CERT\_UNKNOWN** The issuer of the certificate is not recognized or some unspecified issue arose in the processing of the certificate, rendering it unacceptable.

## Compatibility

The ioctl codes with **T == 0** are a subset of the ioctl codes used in Berkeley sockets. In particular, there is no command that 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 can be retrieved by calling **WSAGetLastError**.

## Error Codes

<b>WSAENETDOWN</b>	The network subsystem has failed.
<b>WSAEFAULT</b>	The <i>lpvInBuffer</i> , <i>lpvOutBuffer</i> <i>lpcbBytesReturned</i> , <i>lpOverlapped</i> , or <i>lpCompletionRoutine</i> argument is not totally contained in a valid part of the user address space, or the <i>cbInBuffer</i> or <i>cbOutBuffer</i> argument is too small.
<b>WSAEINVAL</b>	<i>dwIoControlCode</i> 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.
<b>WSAEINPROGRESS</b>	The function is invoked when a callback is in progress.
<b>WSAENOTSOCK</b>	The descriptor <i>s</i> is not a socket.
<b>WSAEOPNOTSUPP</b>	The specified ioctl command cannot be realized. (For example, the <b>FLOWSPEC</b> structures specified in <b>SIO_SET_QOS</b> or <b>SIO_SET_GROUP_QOS</b> cannot be satisfied.)
<b>WSA_IO_PENDING</b>	An overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSAEWOULDBLOCK</b>	The socket is marked as nonblocking and the requested operation would block.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[getsockopt](#), [ioctlsocket](#), [setsockopt](#), [socket](#), [WSASocket](#)

## WSAIsBlocking

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

The Windows Sockets **WSAJoinLeaf** function joins a leaf node into a multipoint session, exchanges connect data, and specifies needed quality of service based on the supplied **FLOWSPEC** structures.

```
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 [FLOWSPEC](#) structures for socket *s*, one for each direction.

*lpGQOS*

[in] Reserved for future use with socket groups. A pointer to the [FLOWSPEC](#) structures for the socket group (if applicable).

*dwFlags*

[in] Flags to indicate that the socket is acting as a sender, receiver, or both.

## Remarks

The **WSAJoinLeaf** 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.

The **WSAJoinLeaf** function 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 can be used for input parameter *s* in this function. The returned socket descriptor will *not* be useable until after the join operation completes. For example, if the socket is in nonblocking mode after a corresponding FD\_CONNECT indication has been received from [WSAAsyncSelect](#) or [WSAEventSelect](#) on the original socket *s*, except that **closesocket** may be invoked on this new socket descriptor to cancel a pending join operation. A root application in a multipoint session may call **WSAJoinLeaf** one or more times in order to add a number of leaf nodes, however at most one multipoint connection request may be outstanding at a time. Refer to [Multipoint and Multicast Semantics](#) for additional information.

For nonblocking sockets it is often not possible to complete the connection immediately. In such a case, this function returns an as-yet unusable socket descriptor and the operation proceeds. There is no error code such as WSAEWOULDBLOCK in this case, since the function has effectively returned a "successful start" indication. When the final outcome success or failure becomes known, it may be reported through [WSAAsyncSelect](#) or [WSAEventSelect](#) depending on how the client registers for notification on the original socket *s*. In either case, the notification is announced with FD\_CONNECT and the error code associated with the FD\_CONNECT indicates either success or a specific reason for failure. The **select** function cannot be used to detect completion notification for **WSAJoinLeaf**.

The socket descriptor returned by **WSAJoinLeaf** is different depending on whether the input socket descriptor, *s*, is a *c\_root* or a *c\_leaf*. When used with a *c\_root* socket, the *name* parameter designates a particular leaf node to be added and the returned socket descriptor is a *c\_leaf* socket corresponding to the newly added leaf node. The newly created socket has the same properties as *s* including asynchronous events registered with [WSAAsyncSelect](#) or with [WSAEventSelect](#), but *not* including the *c\_root* socket's group ID, if any. It is not intended to be used for exchange of multipoint data, but rather is used to receive network event indications (for example, FD\_CLOSE) for the connection that exists to the particular *c\_leaf*. Some multipoint implementations can 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 that 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 that will contain any user data passed back from the peer as part of the multipoint session establishment. The *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 **WSAJoinLeaf** function returns. For nonblocking sockets, this will be after the join operation has completed. For example, this could occur after FD\_CONNECT notification on the original socket *s*). If *lpCalleeData* is NULL, no user data will be passed back. The exact format of the user data is specific to the address family to which the socket belongs.

At multipoint session establishment time, an application can use the *lpSQOS* and/or *lpGQOS* parameters to override any previous quality of service specification made for the socket through **WSAIoctl** with either the SIO\_SET\_QOS or SIO\_SET\_GROUP\_QOS opcodes.

The *lpSQOS* parameter specifies the [FLOWSPEC](#) structures 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 quality of service 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 quality of service.

Reserved for future socket groups. The *lpGQOS* parameter specifies the [FLOWSPEC](#) structures for the socket group (if applicable), one for each direction, followed by any additional provider-specific parameters. If no provider-specific parameters are supplied, the *buf* and *len* fields of *lpGQOS->ProviderSpecific* should be set to NULL and zero, respectively. A NULL value for *lpGQOS* indicates no application-supplied group quality of service. 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).

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** that is a descriptor for the newly created multipoint socket. Otherwise, a value of **INVALID\_SOCKET** is returned, and a

specific error code can 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 on the original socket *s* when the join operation completes, either successfully or otherwise. The application must use either [WSAAsyncSelect](#) or [WSAEventSelect](#) with interest registered for the `FD_CONNECT` event in order to determine when the join operation has completed and checks the associated error code to determine the success or failure of the operation. The `select` function cannot be used to determine when the join operation completes.

Also, until the multipoint session join attempt completes all subsequent calls to **WSAJoinLeaf** on the same socket will fail with the error code `WSAEALREADY`. After the **WSAJoinLeaf** operation completes successfully, a subsequent attempt will usually fail with the error code `WSAEISCONN`. An exception to the `WSAEISCONN` rule occurs for a *c\_root* socket that allows root-initiated joins. In such a case, another join may be initiated after a prior **WSAJoinLeaf** operation completes.

If the return error code indicates the multipoint session join attempt failed (that is, `WSAECONNREFUSED`, `WSAENETUNREACH`, `WSAETIMEDOUT`) the application can call **WSAJoinLeaf** again for the same socket.

## Error Codes

<code>WSANOTINITIALISED</code>	A successful <a href="#">WSAStartup</a> must occur before using this function.
<code>WSAENETDOWN</code>	The network subsystem has failed.
<code>WSAEADDRINUSE</code>	The socket's local address is already in use and the socket was not marked to allow address reuse with <code>SO_REUSEADDR</code> . This error usually occurs at the time of <b>bind</b> , but could be delayed until this function if the <b>bind</b> was to a partially wild-card address (involving <code>ADDR_ANY</code> ) and if a specific address needs to be "committed" at the time of this function.
<code>WSAEINTR</code>	A blocking Windows Socket 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .
<code>WSAEINPROGRESS</code>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<code>WSAEALREADY</code>	A nonblocking <a href="#">WSAJoinLeaf</a> call is in progress on the specified socket.
<code>WSAEADDRNOTAVAIL</code>	The remote address is not a valid address (such as <code>ADDR_ANY</code> ).
<code>WSAEAFNOSUPPORT</code>	Addresses in the specified family cannot be used with this socket.
<code>WSAECONNREFUSED</code>	The attempt to join was forcefully rejected.

WSAEFAULT	The <i>name</i> or the <i>namelen</i> parameter is not a valid part of the user address space, the <i>namelen</i> parameter is too small, the buffer length for <i>lpCalleeData</i> , <i>lpSQOS</i> , and <i>lpGQOS</i> are too small, or the buffer length for <i>lpCallerData</i> 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 <b>FLOWSPEC</b> structures specified in <i>lpSQOS</i> and <i>lpGQOS</i> cannot be satisfied.
WSAEPROTONOSUPPORT	The <i>lpCallerData</i> augment is not supported by the service provider.
WSAETIMEDOUT	Attempt to join timed out without establishing a multipoint session.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[accept](#), [bind](#), [select](#), [WSAAccept](#), [WSAAsyncSelect](#), [WSAEventSelect](#), [WSASocket](#)

## WSALookupServiceBegin

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] A pointer to the search criteria. See below for details.

*dwControlFlags*

[in] A flag that 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_FLUSHPREVIOUS	Used as a value for the <i>dwControlFlags</i> argument in <b>WSALookupServiceNext</b> . Setting this flag instructs the provider to discard the last result set, which was too large for the supplied buffer, and move on to the next result set.
LUP_NEAREST	If possible, return results in the order of distance. The measure of distance is provider specific.
LUP_RES_SERVICE	This indicates whether prime response is in the remote or local part of <b>CSADDR_INFO</b> structure. The other part needs to be "usable" in either case.
LUP_RETURN_ALIASES	Any available alias information is to be returned in successive calls to <b>WSALookupServiceNext</b> , and each alias returned will have the RESULT_IS_ALIAS flag set.
LUP_RETURN_NAME	Retrieve the name as <i>lpServiceInstanceName</i> .
LUP_RETURN_TYPE	Retrieve the type as <i>lpServiceClassId</i> .
LUP_RETURN_VERSION	Retrieve the version as <i>lpVersion</i> .
LUP_RETURN_COMMENT	Retrieve the comment as <i>lpComment</i> .
LUP_RETURN_ADDR	Retrieve the addresses as <i>lpCsaBuffer</i> .
LUP_RETURN_BLOB	Retrieve the private data as <i>lpBlob</i> .
LUP_RETURN_ALL	Retrieve all of the information

*lphLookup*

[out] A 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 can 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 can 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);
```

This call is 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.

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 can be supplied, indicating that the field will *not* be used as a search criteria. See section [Query-Related Data Structures](#) for additional information.

<b>WSAQUERYSET Field Name</b>	<b>Query Interpretation</b>
<i>dwSize</i>	Must be set to sizeof( <b>WSAQUERYSET</b> ). This is a versioning mechanism.
<i>DwOutputflags</i>	Ignored for queries.
<i>LpszServiceInstanceName</i>	<i>(Optional)</i> Referenced string contains service name. The semantics for wildcarding within the string are not defined, but can be supported by certain name space providers.
<i>LpServiceClassId</i>	<i>(Required)</i> The GUID corresponding to the service class.
<i>LpVersion</i>	<i>(Optional)</i> 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).
<i>LpszComment</i>	Ignored for queries.
<i>DwNameSpace1</i>	Identifier of a single name space in which to constrain the search, or NS_ALL to include all name spaces.
<i>LpNSProviderId</i>	<i>(Optional)</i> References the GUID of a specific name space provider, and limits the query to this provider only.
<i>LpszContext</i>	<i>(Optional)</i> Specifies the starting point of the query in a hierarchical name space.
<i>DwNumberOfProtocols</i>	Size of the protocol constraint array, can be zero.

<i>LpafpProtocols</i>	<i>(Optional)</i> References an array of AFPROTOCOLS structure. Only services that utilize these protocols will be returned.
<i>LpszQueryString</i>	<i>(Optional)</i> Some namespaces (such as whois++) support enriched SQL like queries that are contained in a simple text string. This parameter is used to specify that string.
<i>DwNumberOfCsAddrs</i>	Ignored for queries.
<i>LpcsaBuffer</i>	Ignored for queries.
<i>LpBlob</i>	<i>(Optional)</i> 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 can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSAEINVAL	One or more parameters were missing or invalid for this provider.
WSANO_DATA	The name was found in the database but no data matching the given restrictions was located.
WSANOTINITIALIZED	The WS2_32.DLL has not been initialized. The application must first call <a href="#">WSAStartup</a> before calling any Windows Sockets functions.
WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space.
WSA_NOT_ENOUGH_MEMORY	There was insufficient memory to perform the operation.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[WSALookupServiceEnd](#), [WSALookupServiceNext](#)

# WSALookupServiceEnd

The Windows Sockets **WSALookupServiceEnd** function is called to free the handle after previous calls to [WSALookupServiceBegin](#) and [WSALookupServiceNext](#).

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] A 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 can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSA\_INVALID\_HANDLE

The handle is not valid

WSANOTINITIALIZED

The WS2\_32.DLL has not been initialized. The application must first call [WSAStartup](#) before calling any Windows Sockets functions.

WSA\_NOT\_ENOUGH\_MEMORY

There was insufficient memory to perform the operation.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[WSALookupServiceBegin](#), [WSALookupServiceNext](#)

# WSALookupServiceNext

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] A handle returned from the previous call to [WSALookupServiceBegin](#).

### *dwControlFlags*

[in] Flags to control the next operation. Currently only LUP\_FLUSHPREVIOUS is defined as a means to cope with a result set which is too large. If an application does not wish to (or cannot) supply a large enough buffer, setting LUP\_FLUSHPREVIOUS instructs the provider to discard the last result set - which was too large - and move on to the next set for this call.

### *lpdwBufferLength*

[in/out] On input, the number of bytes contained in the buffer pointed to by *lpqsResults*. 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.

## Remarks

The *dwControlFlags* specified in this function and the ones specified at the time of **WSALookupServiceBegin** are treated as "restrictions" for the purpose of combination. The restrictions are combined between the ones at **WSALookupServiceBegin** time and the ones at **WSALookupServiceNext** time. Therefore the flags at **WSALookupServiceNext** can never increase the amount of data returned beyond what was requested at **WSALookupServiceBegin**, although it is NOT an error to specify more or fewer flags. The flags specified at a given **WSALookupServiceNext** apply only to that call.

The *dwControlFlags* LUP\_FLUSHPREVIOUS and LUP\_RES\_SERVICE are exceptions to the "combined restrictions" rule (because they are "behavior" flags instead of "restriction" flags). If either of these flags are used in **WSALookupServiceNext** they have their defined effect regardless of the setting of the same flags at **WSALookupServiceBegin**.

For example, if LUP\_RETURN\_VERSION is specified at **WSALookupServiceBegin** the service

provider retrieves records including the "version". If LUP\_RETURN\_VERSION is NOT specified at **WSALookupServiceNext**, the returned information does not include the "version", even though it was available. No error is generated.

Also for example, if LUP\_RETURN\_BLOB is NOT specified at **WSALookupServiceBegin** but is specified at **WSALookupServiceNext**, the returned information does not include the private data. No error is generated.

## Query Results

The following table describes how the query results are represented in the **WSAQUERYSET** structure.

<b>WSAQUERYSET Field Name</b>	<b>Result Interpretation</b>
<i>dwSize</i>	Will be set to sizeof( <b>WSAQUERYSET</b> ). This is used as a versioning mechanism.
<i>DwOutputFlags</i>	RESULT_IS_ALIAS flag indicates this is an alias result.
<i>LpszServiceInstanceName</i>	Referenced string contains service name.
<i>LpServiceClassId</i>	The GUID corresponding to the service class.
<i>LpVersion</i>	References version number of the particular service instance.
<i>LpszComment</i>	Optional comment string supplied by service instance.
<i>DwNameSpace</i>	Name space in which the service instance was found.
<i>LpNSProviderId</i>	Identifies the specific name space provider that supplied this query result.
<i>LpszContext</i>	Specifies the context point in a hierarchical name space at which the service is located.
<i>DwNumberOfProtocols</i>	Undefined for results.
<i>LpafpProtocols</i>	Undefined for results, all needed protocol information is in the <a href="#">CSADDR_INFO</a> structures.
<i>LpszQueryString</i>	When <i>dwControlFlags</i> includes LUP_RETURN_QUERY_STRING, this field returns the unparsed remainder of the <i>lpszServiceInstanceName</i> specified in the original query. For example, in a name space that identifies services by hierarchical names that specify a host name and a file path within that host, the address returned might be the host address and the unparsed remainder might be the file path. If the <i>lpszServiceInstanceName</i> is fully parsed and LUP_RETURN_QUERY_STRING is used, this field is NULL or points to a zero-length string.
<i>DwNumberOfCsAddrs</i>	Indicates the number of elements in the array of <a href="#">CSADDR_INFO</a> 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 can be retrieved by calling [WSAGetLastError](#).

## Error Codes

`WSA_E_NO_MORE`

There is no more data available. In Windows Sockets version 2, conflicting error codes are defined for `WSAENOMORE` (10102) and `WSA_E_NO_MORE` (10110). The error code `WSAENOMORE` will be removed in a future version and only `WSA_E_NO_MORE` will remain. For Windows Sockets version 2, however, applications should check for both `WSAENOMORE` and `WSA_E_NO_MORE` for the widest possible compatibility with Name Space Providers that use either one.

`WSA_E_CANCELLED`

A call to **WSALookupServiceEnd** was made while this call was still processing. The call has been canceled. The data in the *lpqsResults* buffer is undefined. In Windows Sockets version 2, conflicting error codes are defined for `WSAECANCELLED` (10103) and `WSA_E_CANCELLED` (10111). The error code `WSAECANCELLED` will be removed in a future version and only `WSA_E_CANCELLED` will remain. For Windows Sockets version 2, however, applications should check for both `WSAECANCELLED` and `WSA_E_CANCELLED` for the widest possible compatibility with Name Space Providers that use either one.

`WSAEFAULT`

The *lpqsResults* buffer was too small to contain a [WSAQUERYSET](#) set.

`WSAEINVAL`

One or more required parameters were invalid or missing.

`WSA_INVALID_HANDLE`

The specified Lookup handle is invalid.

`WSANOTINITIALIZED`

The `WS2_32.DLL` has not been initialized. The application must first call [WSAStartup](#) before calling any Windows Sockets functions.

`WSANO_DATA`

The name was found in the database, but no data matching the given restrictions was located.

WSASERVICE_NOT_FOUND	No such service is known. The service cannot be found in the specified name space.
WSA NOT ENOUGH MEMORY	There was insufficient memory to perform the operation.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[WSALookupServiceBegin](#), [WSALookupServiceEnd](#)

# WSANTohl

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

The **WSANTohl** function 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 can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lphostlong</i> parameter is not completely contained in a valid part of the user address space.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[htonl](#), [htons](#), [ntohl](#), [ntohs](#), [WSAHtonl](#), [WSAHtons](#), [WSANtohs](#)

# WSANtohs

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

The **WSANtohs** function 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 can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lphostshort</i> parameter is not completely contained in a valid part of the user address space.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[htonl](#), [htons](#), [ntohl](#), [ntohs](#), [WSAhtonl](#), [WSANtohl](#), [WSAhtons](#)

# WSAProviderConfigChange

The **WSAProviderConfigChange** function notifies the application when the provider configuration is changed.

```
int WINAPI
WSAProviderConfigChange(
    LPHANDLE lpNotificationHandle,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

## Parameters

*lpNotificationHandle*

(in/out) A pointer to notification handle; if the notification handle is set to NULL (the handle value not the pointer itself), this function returns notification handle in the location pointed by *lpNotificationHandle*.

*lpOverlapped*

(in) A pointer to a [WSAOVERLAPPED](#) structure.

*lpCompletionRoutine*

(in) A pointer to the completion routine called when the provider change notification is received.

## Remarks

The **WSAProviderConfigChange** function notifies the application of provider (both transport and name space) installation or removal in Win32 operating environments that support such configuration change without requiring a restart. When called for the first time (*lpNotificationHandle* parameter points to NULL handle), this function completes immediately and returns notification handle in the location pointed by *lpNotificationHandle* that can be used in subsequent calls to receive notifications of provider installation and removal. The second and any subsequent calls only complete when provider information changes since the time the call was made. It is expected (but not required) that that application uses overlapped I/O on second and subsequent calls to **WSAProviderConfigChange**, in which case the call will return immediately and application will be notified of provider configuration changes using the completion mechanism chosen through specified overlapped completion parameters.

Notification handle returned by **WSAProviderConfigChange** is like any regular operating system handle that should be closed (when no longer needed) using Win32 **CloseHandle** call.

The following sequence of actions can be used to guarantee that application always has current protocol configuration information:

- call **WSAProviderConfigChange**
- call [WSAEnumProtocols](#) and/or [WSAEnumNameSpaceProviders](#)
- whenever **WSAProviderConfigChange** notifies application of provider configuration change (via blocking or overlapped IO), the whole sequence of actions should be repeated

## Return Values

If no error occurs the **WSAProviderConfigChange** returns 0. Otherwise, a value of SOCKET\_ERROR is returned and a specific error code may be retrieved by calling **WSAGetLastError**. The error code WSA\_IO\_PENDING indicates that the overlapped operation has been successfully initiated and that completion (and thus change event) will be indicated at a later time

## Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSA_NOT_ENOUGH_MEMORY	Not enough free memory available to complete the operation.
WSA_INVALID_HANDLE	Value pointed by <i>lpNotificationHandle</i> parameter is not a valid notification handle.
WSAEOPNOTSUPP	Current operating system environment does not support provider installation or removal without restart.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws\_32.lib.

## See Also

[WSAEnumProtocols](#), [WSAEnumNameSpaceProviders](#)

# WSARecv

The Windows Sockets **WSARecv** function receives data from a connected 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

The **WSARecv** 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. However, the MSG\_PARTIAL flag

bit is not supported by all protocols.

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

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

For 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 that has already been received and buffered by the transport will be copied into the supplied user buffers. In 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. Windows Socket 2 does not define any standard blocking timeout mechanism for this function. For protocols acting as byte-stream protocols the stack tries to return as much data as possible subject to the supplied buffer space and amount of received data available. However, receipt of a single byte is sufficient to unblock the caller. There is no guarantee that more than a single byte will be returned. For protocols acting as message-oriented, a full message is required to unblock the caller.

Whether or not a protocol is acting as byte-stream is determined by the setting of `XP1_MESSAGE_ORIENTED` and `XP1_PSEUDO_STREAM` in its `WSAPROTOCOL_INFO` structure and the setting of the `MSG_PARTIAL` flag passed in to this function (for protocols that support it). The relevant combinations are summarized in the following table (an asterisk (\*) indicates that the setting of this bit does not matter in this case).

<code>XP1_MESSAGE_ORIENTED</code>	<code>XP1_PSEUDO_STREAM</code>	<code>MSG_PARTIAL</code>	Acts as
not set	*	*	byte-stream
*	set	*	byte-stream
set	not set	set	byte-stream
set	not set	not set	message-oriented

The supplied buffers are filled in the order in which they appear in the array pointed to by *lpBuffers*, and the buffers are packed so that no holes are created.

The array of [WSABUF](#) structures pointed to by the *lpBuffers* parameter is transient. If this operation completes in an overlapped manner, it is the service provider's responsibility to capture

these **WSABUF** structures before returning from this call. This enables applications to build stack-based **WSABUF** arrays.

For byte stream-style sockets (for example, type `SOCK_STREAM`), incoming data is placed into the buffers until the buffers are filled, the connection is closed, or the 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 that depend on whether the socket is a byte stream or message oriented. For byte streams, zero bytes having been read (as indicated by zero return value to indicate success, and *lpNumberOfBytesRecv* value of zero) indicates graceful closure and that no more bytes will ever be read. For message-oriented sockets, where a zero byte message is often allowable, a failure with an error code of `WSAEDISCON` is used to indicate graceful closure. In any case a return error code of `WSAECONNRESET` indicates an abortive close has occurred.

The *lpFlags* 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 *lpFlags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
<code>MSG_PEEK</code>	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.
<code>MSG_OOB</code>	Process out-of-band data. (See section <a href="#">DECnet Out-Of-band data</a> for a discussion of this topic.)
<code>MSG_PARTIAL</code>	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 <code>MSG_PARTIAL</code> 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 indicated by the *lpdwFlags* parameter.

### Overlapped socket I/O

If an overlapped operation completes immediately, **WSARecv** returns a value of zero and the *lpNumberOfBytesRecv* parameter is updated with the number of bytes received and the flag bits indicated 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, *lpNumberOfBytesRecv* 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](#).

The **WSARecv** function can 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. 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 [WSAOVERLAPPED](#) structure.

If the *lpCompletionRoutine* parameter is **NULL**, the *hEvent* field of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use [WSAWaitForMultipleEvents](#) or [WSAGetOverlappedResult](#) to wait or poll on the event object.

If *lpCompletionRoutine* is not **NULL**, the *hEvent* field is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-**NULL** *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped IO request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to **TRUE**. In this case the usage of the **hEvent** field is undefined, and attempting to wait on the *hEvent* field would produce unpredictable results.

The completion routine follows the same rules as stipulated for Win32 file I/O completion routines. The completion routine will *not* be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents** with the *fAlertable* parameter set to **TRUE** is invoked.

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. The *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* parameter specifies the number of bytes received. The *dwFlags* parameter 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 can 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. In this case, the completion routine will have already been scheduled 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 can 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 <a href="#">WSAStartup</a> 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 <a href="#">WSACancelBlockingCall</a> .
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 "keep-alive" activity detecting a failure while the operation was in progress.
WSAENOTSOCK	The descriptor is not a socket.
WSAEFAULT	The <i>lpBuffers</i> parameter is not completely 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 call <a href="#">WSARecv</a> on a socket after <a href="#">shutdown</a> has been invoked with <i>how</i> 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 (for example, with <b>bind</b> ).
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 <i>s</i> 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.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[WSACloseEvent](#), [WSACreateEvent](#), [WSAGetOverlappedResult](#), [WSASocket](#), [WSAWaitForMultipleEvents](#)

## WSARecvDisconnect

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

The **WSARecvDisconnect** function is used on connection-oriented sockets to disable reception and retrieve any incoming disconnect data from the remote party. This is equivalent to a shutdown (SD\_RECV), except that [WSASendDisconnect](#) also allows receipt of disconnect data (in protocols that support it).

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

To successfully receive incoming disconnect data, an application must use other mechanisms to determine that the circuit has been closed. For example, an application needs to receive an FD\_CLOSE notification, to receive a zero return value, or to receive a WSAEDISCON or WSAECONNRESET error code from [recv/WSARecv](#).

The **WSARecvDisconnect** function does not close the socket, and resources attached to the socket will *not* be freed until [closesocket](#) is invoked.

The **WSARecvDisconnect** function 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 disconnected using **WSARecvDisconnect**. 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 can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The buffer referenced by the parameter <i>lpInboundDisconnectData</i> 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.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[connect](#), [socket](#)

# WSARecvEx

**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] An indicator specifying whether the message is fully or partially received for datagram sockets.

## Remarks

The **WSARecvEx** function that is part of the Microsoft implementation of Windows Sockets 2 is similar to the more common [recv](#) function except that the *flags* parameter is an in-out parameter, not just an in parameter. The additional out parameter is used to indicate whether a partial or complete message was received when a message-oriented protocol is being used.

The **WSARecvEx** and [recv](#) function identically for stream oriented protocols.

Making the *flags* parameter an in and out parameter accomodates two common situations in which a partial message will be received: when the application's data buffer size is smaller than the message size and the message coincidentally arrives in two pieces; and when the message is rather large and must arrive in several pieces. The MSG\_PARTIAL bit is set in the *flags* parameter on return from **WSARecvEx** when a partial message was received. If a complete message was received, MSG\_PARTIAL is not set in *flags*.

The Windows Sockets [recv](#) function is different than **WSARecvEx** in that the **recv** function always receives a single message for each call for message-oriented transport protocols. The [recv](#) function does not have a means to indicate to the application that the data received is only a partial message. An application must build its own protocol for checking whether a message is partial or complete by checking for the error code WSAEMSGSIZE after each call to **recv**. When the application buffer is smaller than the data being sent, 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. An application can use **recv** and manage the protocol itself. Alternatively, an applications can use **WSARecvEx** and check the for the MSG\_PARTIAL bit is set in the *flags* parameter.

The **WSARecvEx** function provide the developer with a more effective way of checking whether a message received is partial or complete 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. Instead, **WSARecvEx** can be used, minimizing overhead and eliminating the need for an application-based protocol.

## Return Values

If no error occurs, **WSARecvEx** returns the number of bytes received. If the connection has been closed, it returns zero. Additionally, 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*.

Otherwise, a value of SOCKET\_ERROR is returned, and a specific error code can be retrieved by calling [WSAGetLastError](#).

**Important** For a stream oriented transport protocol, MSG\_PARTIAL is never set on return from **WSARecvEx**. This function behaves identically to the Windows Sockets **recv** function for stream transport protocols.

## Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The <i>buf</i> parameter is not completely contained in a valid part of the user address space.
WSAENOTCONN	The socket is not connected.
WSAEINTR	The (blocking) call was canceled through <a href="#">WSACancelBlockingCall</a> .
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 use <a href="#">WSARecvEx</a> on a socket after <a href="#">shutdown</a> has been invoked with <i>how</i> set to SD_RECEIVE or SD_BOTH.
WSAEWOULDBLOCK	The socket is marked as nonblocking and the receive operation would block.
WSAEINVAL	The socket has not been bound with <a href="#">bind</a> , or an unknown flag was specified, or MSG_OOB was specified for a socket with SO_OOBINLINE enabled or (for byte stream sockets only) <i>len</i> 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.

## QuickInfo

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in mswsock.h.

**Import Library:** Link with mswsock.lib.

## See Also

[recvfrom](#), [select](#), [send](#), [socket](#), [WSAAsyncSelect](#)

# WSARecvFrom

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 that 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

The **WSARecvFrom** 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.

The **WSARecvFrom** function is used primarily on a connectionless socket specified by *s*. The socket's local address must be known. For server applications, this is usually done explicitly through **bind**. Explicit binding is discouraged for client applications. For client applications using this function the socket can become bound implicitly to a local address through [sendto](#), [WSASendTo](#), or [WSAJoinLeaf](#).

For overlapped sockets, this function is used to post one or more buffers into which incoming data will be placed as it becomes available on a (possibly connected) socket, after which the application-specified completion indication (invocation of the completion routine or setting of an event object) occurs. If the operation does not complete immediately, the final completion status is retrieved through the completion routine or [WSAGetOverlappedResult](#). Also, the values indicated by *lpFrom* and *lpFromlen* are not updated until completion is itself 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 [WSARecv](#) function and the *lpOverlapped* and *lpCompletionRoutine* parameters are ignored. Any data that 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 indicated 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 indicated 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, until the connection is closed, or until the 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`.

The *lpFlags* 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 *lpFlags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
<code>MSG_PEEK</code>	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.
<code>MSG_OOB</code>	Process out-of-band data. (See section <a href="#">DECnet Out-Of-band data</a> for a discussion of this topic.)
<code>MSG_PARTIAL</code>	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 <code>MSG_PARTIAL</code> flag cleared indicates end of sender's message.
	As an input parameter indicates that the receive operation should complete even if only part of a message has been received by the service provider.

For message-oriented sockets, the `MSG_PARTIAL` bit is set in the *lpFlags* parameter if a partial message is received. If a complete message is received, `MSG_PARTIAL` is cleared in *lpFlags*. In the case of delayed completion, the value pointed to by *lpFlags* is not updated. When completion has been indicated the application should call **WSAGetOverlappedResult** and examine the flags pointed to by the *lpdwFlags* parameter.

### Overlapped socket I/O

If an overlapped operation completes immediately, **WSARecvFrom** returns a value of zero and the *lpNumberOfBytesRecv* parameter is updated with the number of bytes received and the flag bits pointed by the *lpFlags* parameter are also updated. If the overlapped operation is successfully initiated and will complete later, **WSARecvFrom** returns `SOCKET_ERROR` and indicates error code `WSA_IO_PENDING`. In this case, *lpNumberOfBytesRecv* 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](#).

The **WSARecvFrom** function can 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 [WSAOVERLAPPED](#) structure.

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* field of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use [WSAWaitForMultipleEvents](#) or [WSAGetOverlappedResult](#) to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* field is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-NULL *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped IO request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to TRUE. In this case the usage of the *hEvent* field is undefined, and attempting to wait on the *hEvent* field would produce unpredictable results.

The completion routine follows the same rules as stipulated for Win32 file I/O completion routines. The completion routine will *not* be invoked until the thread is in an alertable wait state such as can occur when the function [WSAWaitForMultipleEvents](#) with the *fAlertable* parameter set to TRUE is invoked.

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  
);
```

The **CompletionRoutine** is a placeholder for an application-defined or library-defined function name. The *dwError* specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* specifies the number of bytes received. The *dwFlags* parameter 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 can 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. In this case, the completion routine will have already been scheduled 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 can 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 <b>WSAStartup</b> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAEFAULT	The <i>lpBuffers</i> , <i>lpFlags</i> , <i>lpFrom</i> , <i>lpNumberOfBytesRecv</i> , <i>lpFromlen</i> , <i>lpOverlapped</i> , or <i>lpCompletionRoutine</i> argument is not totally contained in a valid part of the user address space: the <i>lpFrom</i> buffer was too small to accommodate the peer address.
WSAEINTR	A blocking Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .
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 <b>bind</b> , for example).
WSAEISCONN	The socket is connected. This function is not permitted with a connected socket, whether the socket is connection-oriented or connectionless.
WSAENETRESET	The connection has been broken due to "keep-alive" activity detecting a failure while the operation was in progress.
WSAENOTCONN	The socket is not connected (connection-oriented sockets only).
WSAEOPNOTSUPP	<code>MSG_OOB</code> was specified, but the socket is not stream-style such as type <code>SOCK_STREAM</code> , 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 <b>WSARecvFrom</b> on a socket after <b>shutdown</b> has been invoked with <i>how</i> set to <code>SD_RECEIVE</code> or <code>SD_BOTH</code> .

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.
WSAECONNRESET	The virtual circuit was reset by the remote side executing a "hard" or "abortive" close. The application should close the socket as it is no longer useable. On a UDP datagram socket this error would indicate that a previous send operation resulted in an ICMP "Port Unreachable" message.
WSAEDISCON	Socket <i>s</i> 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.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[WSACloseEvent](#), [WSACreateEvent](#), [WSAGetOverlappedResult](#), [WSASocket](#), [WSAWaitForMultipleEvents](#)

## WSARemoveServiceClass

The Windows Sockets **WSARemoveServiceClass** function permanently unregisters service class schema.

```
INT WSARemoveServiceClass(  
    LPGUID lpServiceClassId  
);
```

### Parameters

*lpServiceClassId*

[in] A 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 can be retrieved by calling [WSAGetLastError](#).

## Error Codes

<code>WSATYPE_NOT_FOUND</code>	The specified class was not found.
<code>WSAEACCES</code>	The calling routine does not have sufficient privileges to remove the Service.
<code>WSANOTINITIALIZED</code>	The <code>WS2_32.DLL</code> has not been initialized. The application must first call <a href="#">WSAStartup</a> before calling any Windows Sockets functions.
<code>WSAEINVAL</code>	The specified GUID was not valid.
<code>WSA NOT ENOUGH MEMORY</code>	There was insufficient memory to perform the operation

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in `wsocket2.h`.

**Import Library:** Link with `ws2_32.lib`.

# WSAResetEvent

The Windows Sockets **WSAResetEvent** function resets the state of the specified event object to nonsignaled.

```
BOOL WSAResetEvent(  
    WSAEVENT hEvent  
);
```

## Parameters

*hEvent*

[in] A handle that identifies an open event object handle.

## Remarks

The **WSAResetEvent** function is used to set the state of the event object to nonsignaled.

## Return Values

If the **WSAResetEvent** 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 <a href="#">WSAStartup</a> 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	The <i>hEvent</i> parameter is not a valid event object handle.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[WSACloseEvent](#), [WSACreateEvent](#), [WSASetEvent](#)

## WSASend

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] A flag that 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

The **WSASend** 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.

The **WSASend** function is used to write outgoing data from one or more buffers on a connection-oriented socket specified by *s*. It can also be used, however, on connectionless sockets that have a stipulated default peer address established through the [connect](#) or [WSAConnect](#) function.

For overlapped sockets (created using [WSASocket](#) with flag `WSA_FLAG_OVERLAPPED`) sending information uses overlapped I/O, unless both *lpOverlapped* and *lpCompletionRoutine* are NULL. In that case, the socket is treated as a nonoverlapped socket. A completion indication will occur, invoking 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](#).

If both *lpOverlapped* and *lpCompletionRoutine* are NULL, the socket in this function will be treated as a non-overlapped socket.

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** The successful completion of a **WSASend** does not indicate that the data was successfully delivered.

The *dwFlags* 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 *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 <a href="#">DECnet Out-Of-band data</a> for a discussion of this topic.)
MSG_PARTIAL	Specifies that <i>lpBuffers</i> only contains a partial message. Note that the error code WSAEOPNOTSUPP will be returned by transports that 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](#).

The **WSASend** function can 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 [WSAOVERLAPPED](#) structure.

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* field of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use **WSAWaitForMultipleEvents** or **WSAGetOverlappedResult** to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* field is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-NULL *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped IO request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to TRUE. In this case the usage of the *hEvent* field is undefined, and attempting to wait on the *hEvent* field would produce unpredictable results.

The completion routine follows the same rules as stipulated for Win32 file I/O completion

routines. The completion routine will *not* be invoked until the thread is in an alertable wait state such as can occur when the function **WSAWaitForMultipleEvents** with the *fAlertable* parameter set to TRUE is invoked.

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
);
```

The **CompletionRoutine** function 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 can 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. In this case, the completion routine will have already been scheduled 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 can 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 <b>WSAStartup</b> 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	A blocking Windows Socket 1.1 call was canceled through <b>WSACancelBlockingCall</b> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.

WSAEFAULT	The <i>lpBuffers</i> , <i>lpNumberOfBytesSent</i> , <i>lpOverlapped</i> , <i>lpCompletionRoutine</i> argument is not totally contained in a valid part of the user address space.
WSAENETRESET	The connection has been broken due to "keep-alive" activity detecting a failure while the operation was in progress.
WSAENOBUFS	The 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 <a href="#">WSASend</a> on a socket after <a href="#">shutdown</a> 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 <a href="#">bind</a> , 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 <a href="#">WSAIoctl</a> .

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[WSACloseEvent](#), [WSACreateEvent](#), [WSAGetOverlappedResult](#), [WSASocket](#), [WSAWaitForMultipleEvents](#)

## WSASendDisconnect

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

The **WSASendDisconnect** functions is used on connection-oriented sockets to disable transmission, and to initiate termination of the connection along with the transmission of disconnect data, if any. This is equivalent to a shutdown(SD\_SEND), except that **WSASendDisconnect** also allows sending disconnect data (in protocols that support it).

After this function has been successfully issued, subsequent sends are disallowed.

The *lpOutboundDisconnectData* parameter, if not NULL, points to a buffer containing the outgoing disconnect data to be sent to the remote party for retrieval by using [WSARecvDisconnect](#).

The **WSASendDisconnect** function does not close the socket, and resources attached to the socket will *not* be freed until [closesocket](#) is invoked.

The **WSASendDisconnect** function 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 calling **WSASendDisconnect**. 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 can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> must occur before using this function.
WSAENETDOWN	The network subsystem has failed.
WSAENOPROTOOPT	The parameter <i>lpOutboundDisconnectData</i> 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 <i>lpOutboundDisconnectData</i> parameter is not completely contained in a valid part of the user address space.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[connect](#), [socket](#)

# WSASendTo

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 (possibly 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] An indicator specifying 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

The **WSASendTo** 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.

The **WSASendTo** function is normally used on a connectionless socket specified by *s* to send a datagram contained in one or more buffers to a specific peer socket identified by the *lpTo* parameter. Even if the connectionless socket has been previously connected using the [connect](#) function to a specific address, *lpTo* overrides the destination address for that particular datagram only. On a connection-oriented socket, the *lpTo* and *iToLen* parameters are ignored; in this case, the **WSASendTo** is equivalent to [WSASend](#).

For overlapped sockets (created using **WSASocket** with flag `WSA_FLAG_OVERLAPPED`) sending data uses 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 (invoking 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](#).

If both *lpOverlapped* and *lpCompletionRoutine* are NULL, the socket in this function will be treated as a non-overlapped socket.

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** returns 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 indicated 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 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.

The successful completion of a **WSASendTo** does not indicate that the data was successfully delivered.

The *dwFlags* 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 *dwFlags* parameter. The latter is constructed by or-ing any of the following values:

Value	Meaning
<code>MSG_DONTROUTE</code>	Specifies that the data should not be subject to routing. A Windows Socket service provider may choose to ignore this flag.
<code>MSG_OOB</code>	Send out-of-band data (stream-style socket such as <code>SOCK_STREAM</code> only).
<code>MSG_PARTIAL</code>	Specifies that <i>lpBuffers</i> only contains a partial message. Note that the error code <code>WSAEOPNOTSUPP</code> will be returned by transports that 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**.

The **WSASendTo** function can 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 [WSAOVERLAPPED](#) structure.

If the *lpCompletionRoutine* parameter is NULL, the *hEvent* field of *lpOverlapped* is signaled when the overlapped operation completes if it contains a valid event object handle. An application can use [WSAWaitForMultipleEvents](#) or [WSAGetOverlappedResult](#) to wait or poll on the event object.

If *lpCompletionRoutine* is not NULL, the *hEvent* field is ignored and can be used by the application to pass context information to the completion routine. A caller that passes a non-NULL *lpCompletionRoutine* and later calls **WSAGetOverlappedResult** for the same overlapped IO request may not set the *fWait* parameter for that invocation of **WSAGetOverlappedResult** to TRUE. In this case the usage of the *hEvent* field is undefined, and attempting to wait on the *hEvent* field would produce unpredictable results.

The completion routine follows the same rules as stipulated for Win32 file I/O completion routines. The completion routine will *not* be invoked until the thread is in an alertable wait state such as can occur when the function [WSAWaitForMultipleEvents](#) with the *fAlertable* parameter set to TRUE is invoked.

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

The prototype of the completion routine is as follows:

```
void CALLBACK CompletionROUTINE(  
    IN DWORD dwError,  
    IN DWORD cbTransferred,  
    IN LPWSAOVERLAPPED lpOverlapped,  
    IN DWORD dwFlags  
);
```

The **CompletionRoutine** function is a placeholder for an application-defined or library-defined function name. The *dwError* parameter specifies the completion status for the overlapped operation as indicated by *lpOverlapped*. The *cbTransferred* parameter 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 can 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. In this case, the completion routine will have already been scheduled 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 can 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 <a href="#">WSAStartup</a> 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	A blocking Windows Socket 1.1 call was canceled through <a href="#">WSACancelBlockingCall</a> .
WSAEINPROGRESS	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
WSAEFAULT	The <i>lpBuffers</i> , <i>lpTo</i> , <i>lpOverlapped</i> , <i>lpNumberOfBytesSent</i> , or <i>lpCompletionRoutine</i> parameters are not part of the user address space, or the <i>lpTo</i> argument is too small.
WSAENETRESET	The connection has been broken due to "keep-alive" activity detecting a failure while the operation was in progress.
WSAENOBUFS	The 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 <b>WSASendTo</b> on a socket after <a href="#">shutdown</a> 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 <b>bind</b> , 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 remote address is not a valid address (such as ADDR_ANY).
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 <a href="#">WSAIoctl</a> .

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[WSACloseEvent](#), [WSACreateEvent](#), [WSAGetOverlappedResult](#), [WSASocket](#), [WSAWaitForMultipleEvents](#)

## WSASetBlockingHook

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.

[WSAIsBlocking](#), [WSACancelBlockingCall](#)

## WSASetEvent

The Windows Sockets **WSASetEvent** function sets the state of the specified event object to signaled.

```
BOOL WSASetEvent (  
    WSAEVENT hEvent  
);
```

### Parameters

*hEvent*

[in] A handle that identifies an open event object.

### Remarks

The **WSASetEvent** function sets the state of the event object 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 <a href="#">WSAStartup</a> 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	The <i>hEvent</i> parameter is not a valid event object handle.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[WSACloseEvent](#), [WSACreateEvent](#), [WSAResetEvent](#)

## WSASetLastError

The Windows Sockets **WSASetLastError** function sets the error code that can be retrieved through the [WSAGetLastError](#) function.

```
void WSASetLastError (  
    int iError  
);
```

### Parameters

*iError*

[in] An integer that specifies the error code to be returned by a subsequent [WSAGetLastError](#) call.

### Remarks

The **WSASetLastError** 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 calling the function [getsockopt](#) with SO\_ERROR.

### Return Values

None.

### Error Codes

WSANOTINITIALISED

A successful [WSAStartup](#) must occur before using this function.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[getsockopt](#), [WSAGetLastError](#)

## WSASetService

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,
    WSAESETSERVICEOP essOperation,
    DWORD dwControlFlags
);
    
```

**Parameters**

*lpqsRegInfo*

[in] A pointer to the service information for registration or 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 must 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 following values:

Flag	Meaning
SERVICE_MULTIPLE	Controls scope of operation. When clear, service addresses are managed as a group. A register or deregister invalidates all existing addresses before adding the given address set. When set, the action is only performed on the given address set. A register does not invalidate existing addresses and a deregister only invalidates the given set of addresses.

The available values for *essOperation* and *dwControlFlags* combine to give meanings as shown in the following table:

Operation	Flags	Service already exists	Service does not exist
RNRSERVICE_REGISTER	none	Overwrite the object. Use only addresses specified. Object is REGISTERED.	Create a new object. Use only addresses specified. Object is REGISTERED.
RNRSERVICE_REGISTER	SERVICE_MULTIPLE	Update object. Add new addresses to existing set. Object is REGISTERED.	Create a new object. Use all addresses specified. Object is REGISTERED.

RNRSERVICE _DEREGISTER	none	Remove all addresses, but do not remove object from name space. Object is DEREGISTERED.	WSASERVICE _NOT_FOUND
RNRSERVICE _DEREGISTER	SERVICE _MULTIPLE	Update object. Remove only addresses that are specified. Only mark object as DEREGISTERED if no addresses present. Do not remove from the name space.	WSASERVICE _NOT_FOUND
RNRSERVICE _DELETE	none	Remove object from the name space.	WSASERVICE _NOT_FOUND
RNRSERVICE _DELETE	SERVICE _MULTIPLE	Remove only addresses that are specified. Only remove object from the name space if no addresses remain.	WSASERVICE _NOT_FOUND

## Remarks

SERVICE\_MULTIPLE lets an application manage its addresses independently. This is useful when the application wants to manage its protocols individually or when the service resides on more than one machine. For instance, when a service uses more than one protocol, it may find that one listening socket aborts but the others remain operational. In this case, the service could deregister the aborted address without affecting the other addresses.

When using SERVICE\_MULTIPLE, an application must not let stale addresses remain in the object. This can happen if the application aborts without issuing a DEREGISTER request. When a service registers, it should store its addresses. On its next invocation, the service should explicitly deregister these old stale addresses before registering new addresses.

## Service Properties

The following table describes how service property data is represented in a **WSAQUERYSET** structure. Fields labeled as *(Optional)* can be supplied with a NULL pointer.

<b>WSAQUERYSET Field Name</b>	<b>Service Property Description</b>
Field Name	Service Property Description
<i>dwSize</i>	Must be set to sizeof( <b>WSAQUERYSET</b> ). This is a versioning mechanism.
<i>DwOutputFlags</i>	Not applicable and ignored.
<i>LpszServiceInstanceName</i>	Referenced string contains the service instance name.
<i>LpServiceClassId</i>	The GUID corresponding to this service class.
<i>lpVersion</i>	<i>(Optional)</i> Supplies service instance version number.
<i>LpszComment</i>	<i>(Optional)</i> An optional comment string.

<i>DwNameSpace</i>	See table below.
<i>LpNSProviderId</i>	See table below.
<i>LpszContext</i>	(Optional) Specifies the starting point of the query in a hierarchical name space.
<i>DwNumberOfProtocols</i>	Ignored.
<i>LpafpProtocols</i>	Ignored.
<i>LpszQueryString</i>	Ignored.
<i>DwNumberOfCsAddrs</i>	The number of elements in the array of CSADDRO_INFO structures referenced by <i>lpcsaBuffer</i> .
<i>LpcsaBuffer</i>	A pointer to an array of CSADDRO_INFO structures that contain the address(es) that the service is listening on.
<i>lpBlob</i>	(Optional) This is a pointer to a provider-specific entity.

As illustrated below, the combination of the *dwNameSpace* and *lpNSProviderId* parameters determine that name space providers are affected by this function.

<i>dwNameSpace</i>	<i>lpNSProviderId</i>	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 for **WSASetService** 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

WSAEACCES	The calling routine does not have sufficient privileges to install the Service.
WSAEINVAL	One or more required parameters were invalid or missing.
WSANOTINITIALIZED	The WiS2_32.DLL has not been initialized. The application must first call <a href="#">WSAStartup</a> before calling any Windows Sockets functions
WSA NOT ENOUGH MEMORY	There was insufficient memory to perform the operation.
WSASERVICE NOT FOUND	No such service is known. The service cannot be found in the specified name space.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## WSASocket

The Windows Sockets **WSASocket** function creates a socket that 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 that 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] Reserved for future use with socket groups. The identifier of the socket group.

*dwFlags*

[in] A flag that specifies the socket attribute.

### Remarks

The **WSASocket** 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 WS2\_32.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 can 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.

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

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

Reserved for future use with socket groups. Parameter *g* is used to indicate the appropriate actions on socket groups:

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 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.

The *dwFlags* parameter can be used to specify the attributes of the socket by or-ing any of the following Flags:

<b>Flag</b>	<b>Meaning</b>
WSA_FLAG_OVERLAPPED	This flag causes an overlapped socket to be created. Overlapped sockets can utilize <a href="#">WSASend</a> , <a href="#">WSASendTo</a> , <a href="#">WSARecv</a> , <a href="#">WSARecvFrom</a> and <a href="#">WSAIoctl</a> for overlapped I/O operations, which allows multiple these operations to be initiated and in progress simultaneously. All functions that allow overlapped operation ( <a href="#">WSASend</a> , <a href="#">WSARecv</a> , <a href="#">WSASendTo</a> , <a href="#">WSARecvFrom</a> , <a href="#">WSAIoctl</a> ) also support non-overlapped usage on an overlapped socket if the values for parameters related to overlapped operation are NULL.
WSA_FLAG_MULTIPOINT_C_ROOT	Indicates that the socket created will be a <i>c_root</i> in a multipoint session. Only allowed if a rooted control plane is indicated in the protocol's <a href="#">WSAPROTOCOL_INFO</a> structure. Refer to <a href="#">Multipoint and Multicast Semantics</a> for additional information.

WSA_FLAG_MULTIPPOINT_C_LEAF	Indicates that the socket created will be a c_leaf in a multicast session. Only allowed if XP1_SUPPORT_MULTIPPOINT is indicated in the protocol's <a href="#">WSAPROTOCOL_INFO</a> structure. Refer to <a href="#">Multipoint and Multicast Semantics</a> for additional information.
WSA_FLAG_MULTIPPOINT_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 <a href="#">WSAPROTOCOL_INFO</a> structure. Refer to <a href="#">Multipoint and Multicast Semantics</a> for additional information.
WSA_FLAG_MULTIPPOINT_D_LEAF	Indicates that the socket created will be a d_leaf in a multipoint session. Only allowed if XP1_SUPPORT_MULTIPPOINT is indicated in the protocol's <a href="#">WSAPROTOCOL_INFO</a> structure. Refer to <a href="#">Multipoint and Multicast Semantics</a> for additional information.

**Important** For multipoint sockets, exactly one of WSA\_FLAG\_MULTIPPOINT\_C\_ROOT or WSA\_FLAG\_MULTIPPOINT\_C\_LEAF *must* be specified, and exactly one of WSA\_FLAG\_MULTIPPOINT\_D\_ROOT or WSA\_FLAG\_MULTIPPOINT\_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 can be sent or received on them. A connection to another socket is created with a [connect/WSAConnect](#) call. Once connected, data can 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 connected to a specific peer, datagrams can be sent to that peer using [send/WSASend](#) and can be received from (only) this peer using [recv/WSARecv](#).

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

### Shared Sockets

When a special [WSAPROTOCOL\\_INFO](#) 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*. Such a [WSAPROTOCOL\\_INFO](#) struct may only be used once, otherwise the error code WSAEINVAL will result.

### 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 can be retrieved by calling [WSAGetLastError](#).

## Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> 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 <i>g</i> specified is not valid, or the <b>WSAPROTOCOL_INFO</b> structure that <i>lpProtocolInfo</i> points to is incomplete, the contents are invalid or the <b>WSAPROTOCOL_INFO</b> structure has already been used in an earlier duplicate socket operation.
WSAEFAULT	<i>lpProtocolInfo</i> argument is not in a valid part of the process address space.
WSAINVALIDPROVIDER	The service provider returned a version other than 2.2.
WSAINVALIDPROCTABLE	The service provider returned an invalid or incomplete procedure table to the <code>WSPStartup</code> .

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in `wsocket2.h`.

**Import Library:** Link with `ws2_32.lib`.

## See Also

[accept](#), [bind](#), [connect](#), [getsockname](#), [getsockopt](#), [ioctlsocket](#), [listen](#), [recv](#), [recvfrom](#), [select](#), [send](#), [sendto](#), [setsockopt](#), [shutdown](#)

# WSAStartup

The Windows Sockets **WSAStartup** function initiates use of WS2\_32.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

The **WSAStartup** 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 can only issue further Windows Sockets functions after a successfully calling **WSAStartup**.

In order to support future Windows Sockets implementations and applications that can have functionality differences from current version of Windows Sockets, a negotiation takes place in **WSAStartup**. The caller of **WSAStartup** and the WS2\_32.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 WS2\_32.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 WS2\_32.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 WS2\_32.DLL or fail to initialize.

It is legal and possible for an application written to this version of the specification to successfully negotiate a higher version number than the version of this specification. In such a case, the application is only guaranteed access to higher-version functionality that fits within the syntax defined in this version, such as new Ioctl codes and new behavior of existing functions. New functions, for example, may be inaccessible. To be guaranteed full access to new syntax of a future version, the application must fully conform to that future version, such as compiling against a new header file, linking to a new library, or other special cases.

This negotiation allows both a WS2\_32.DLL and a Windows Sockets application to support a range of Windows Sockets versions. An application can use WS2\_32.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 WS2\_32.DLL versions:

<b>App versions</b>	<b>DLL Versions</b>	<b>wVersion Requested</b>	<b>wVersion</b>	<b>wHigh Version</b>	<b>End Result</b>
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 that supports only version 2.2 of Windows Sockets makes a **WSAStartup** call:

```
WORD wVersionRequested;
WSADATA wsaData;
int err;

wVersionRequested = MAKEWORD( 2, 2 );

err = WSAStartup( wVersionRequested, &wsaData );
if ( err != 0 ) {
    /* Tell the user that we could not find a usable */
    /* WinSock DLL. */
    return;
}

/* Confirm that the WinSock DLL supports 2.2.*/
/* Note that if the DLL supports versions greater */
/* than 2.2 in addition to 2.2, it will still return */
/* 2.2 in wVersion since that is the version we */
/* requested. */

if ( LOBYTE( wsaData.wVersion ) != 2 ||
      HIBYTE( wsaData.wVersion ) != 2 ) {
    /* Tell the user that we could not 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 can proceed to make other Windows Sockets calls as needed. When it has finished using the services of the WS2\_32.DLL, the application or DLL must call [WSACleanup](#) in order to allow the WS2\_32.DLL to free any resources for the application.

Details of the actual Windows Sockets implementation are described in the [WSADATA](#) structure.

An application or DLL can call **WSAStartup** more than once if it needs to obtain the WSADATA

structure information more than once. On each such call the application can specify any version number supported by the DLL.

An application must call one **WSACleanup** call for every successful **WSAStartup** call to allow third-party DLLs to make use of a WS2\_32.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

The **WSAStartup** function returns zero if successful. Otherwise, it returns one of the error codes listed below.

An application cannot call [WSAGetLastError](#) to determine the error code as is normally done in Windows Sockets if **WSAStartup** fails. The WS2\_32.DLL will *not* have been loaded in the case of a failure so the client data area where the "last error" information is stored could not be established.

### 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 <i>lpWSAData</i> is not a valid pointer.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Use version 1.0 and later.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

### See Also

[send](#), [sendto](#), [WSACleanup](#)

## WSAStringToAddress

The Windows Sockets **WSAStringToAddress** function converts a numeric string to a [SOCKADDR](#) structure, suitable for passing to Windows Sockets routines that take such a

structure.

```
INT WSAStringToAddress(  
    LPTSTR AddressString,  
    INT AddressFamily,  
    LPWSAPROTOCOL_INFO lpProtocolInfo,  
    LP SOCKADDR lpAddress,  
    LPINT lpAddressLength  
);
```

## Parameters

### *AddressString*

[in] Points to the zero-terminated human-readable numeric string to convert.

### *AddressFamily*

[in] The address family to which the string belongs.

### *lpProtocolInfo*

[in] (optional) the [WSAPROTOCOL\\_INFO](#) structure associated with the provider to be used. If this is NULL, the call is routed to the provider of the first protocol supporting the indicated AddressFamily.

### *Address*

[out] A buffer that is filled with a single [SOCKADDR](#).

### *lpAddressLength*

[in/out] The length of the Address buffer. Returns the size of the resultant [SOCKADDR](#) structure. If the supplied buffer is not large enough, the function fails with a specific error of WSAEFAULT and this parameter is updated with the required size in bytes.

## Remarks

The **WSAStringToAddress** function converts alpha-numeric address to [SOCKADDR](#) structures. WSAStringToAddress is the protocol independent equivalent of the BSD **inet\_ntoa** function.

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.

## Return Values

The return value for **WSAStringToAddress** 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 Address buffer is too small. Pass in a larger buffer.
WSAEINVAL	Unable to translate the string into a <a href="#">SOCKADDR</a> .
WSANOTINITIALIZED	The WS2_32.DLL has not been initialized. The application must first call <a href="#">WSAStartup</a> before calling any Windows Socket functions.

WSA NOT ENOUGH MEMORY

There was insufficient memory to perform the operation.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

# WSAUnhookBlockingHook

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

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] An indicator specifying 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] A pointer to an array of event object handles.

*fWaitAll*

[in] An indicator specifying 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] An indicator specifying 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] An indicator specifying 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 *fAlertable* 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 **WSAWaitForMultipleEvents** 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 <i>fWaitAll</i> is TRUE, the return value indicates that the state of all specified event objects is signaled. If <i>fWaitAll</i> is FALSE, the return value minus WSA_WAIT_EVENT_0 indicates the <i>lphEvents</i> 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 <i>fWaitAll</i> parameter are not satisfied.

## Error Codes

WSANOTINITIALISED	A successful <a href="#">WSAStartup</a> 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 <i>lphEvents</i> array is not a valid event object handle.
WSA_INVALID_PARAMETER	The <i>cEvents</i> parameter does not contain a valid handle count.

## QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock2.h.

**Import Library:** Link with ws2\_32.lib.

## See Also

[WSACloseEvent](#), [WSACreateEvent](#)

# AFPROTOCOLS

```
typedef struct _AFPROTOCOLS {
    INT iAddressFamily;
    INT iProtocol;
} AFPROTOCOLS, *PAFPROTOCOLS, *LPAFPROTOCOLS;
```

# BLOB

A **BLOB** structure contains information about a block of data.

```
typedef struct _BLOB {
    ULONG    cbSize;
    BYTE    *pBlobData;
} BLOB;
```

## Members

**cbSize**

Specifies the size in bytes of the block of data pointed to by **pBlobData**

**Addresses**

Points to a block of data.

**Remarks**

The structure name **BLOB** comes from the acronym BLOB, which stands for "Binary Large Object".

Note that this structure description says nothing about the nature of the data pointed to by **pBlobData**.

**QuickInfo**

**Windows CE:** Unsupported.

**Header:** Declared in wtypes.h.

**See Also**[SERVICE\\_INFO](#)

## CSADDR\_INFO

The **CSADDR\_INFO** structure contains Windows Sockets address information for a network service or name space provider. The [GetAddressByName](#) function obtains Windows Sockets address information using **CSADDR\_INFO** structures.

```
typedef struct _CSADDR_INFO {
    SOCKET_ADDRESS LocalAddr;
    SOCKET_ADDRESS RemoteAddr;
    INT             iSocketType;
    INT             iProtocol;
} CSADDR_INFO;
```

**Members****LocalAddr**

Specifies a Windows Sockets local address.

In a client application, pass this address to the [bind](#) function to obtain access to a network service.

In a network service, pass this address to the **bind** function so that the service is bound to the appropriate local address.

**RemoteAddr**

Specifies a Windows Sockets remote address. There are several uses for this remote address:

- You can use this remote address to connect to the service via the [connect](#) function.

This is useful if an application performs [send/receive](#) operations that involve connection-oriented protocols.

- You can use this remote address with the [sendto](#) function when you are communicating over a connectionless (datagram) protocol. If you are using a connectionless protocol, such as UDP, [sendto](#) is typically the way you pass data to the remote system.

### iSocketType

Specifies the type of the Windows socket. The following socket types are defined in WINSOCK.H:

Value	Socket Type
SOCK_STREAM	Stream. This is a protocol that sends data as a stream of bytes, with no message boundaries.
SOCK_DGRAM	Datagram. This is a connectionless protocol. There is no virtual circuit setup. There are typically no reliability guarantees. Services use <b>recvfrom</b> to obtain datagrams. The <b>listen</b> and <b>accept</b> functions do not work with datagrams.
SOCK_RDM	Reliably-Delivered Message. This is a protocol that preserves message boundaries in data.
SOCK_SEQPACKET	Sequenced packet stream. This is a protocol that is essentially the same as SOCK_RDM.

### iProtocol

Specifies a value to pass as the protocol parameter to the **socket** function to open a socket for this service.

### QuickInfo

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in nspapi.h.

### See Also

[bind](#), [connect](#), [GetAddressByName](#), [recv](#), [send](#), [sendto](#)

## 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;
```

### Element

fd\_count

fd\_array

### Usage

The number of sockets that are set.

An array of the sockets in the set.

# FLOWSPEC

```

typedef struct _flowspec {
    uint32      TokenRate;           /* In Bytes/sec */
    uint32      TokenBucketSize;    /* In Bytes */
    uint32      PeakBandwidth;      /* In Bytes/sec */
    uint32      Latency;            /* In microseconds */
    uint32      DelayVariation;     /* In microseconds */
    SERVICE_TYPE ServiceType;       /* Guaranteed, Predictive, */
                                        /* Best Effort, etc. */
    uint32      MaxSduSize          /* In Bytes */
    uint32      MinimumPolicedSize /* In Bytes */
} FLOWSPEC, *PFLOWSPEC, FAR * LPFLOWSPEC;

typedef struct _QualityOfService {
    FLOWSPEC SendingFlowspec /*The flowspec for data sending*/
    FLOWSPEC ReceivingFlowspec /*Theflowspec for data receiving*/
    WSABUF ProviderSpecific /*Additonal Provider*/
} QOS, FAR * LPQOS;

```

## 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

Largest typical frame size in video applications, expressed in bytes. In constant rate applications, the **TokenBucketSize** is chosen to accommodate small variations.

### PeakBandwidth

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

### Latency

Maximum acceptable delay between transmission of a bit by the sender and its receipt by the intended receiver or receivers, expressed in microseconds. The precise interpretation of this number depends on the level of guarantee specified in the QOS request.

### DelayVariation

Difference, in microseconds, between the maximum and minimum possible delay that a packet will experience. This value is used by applications to determine the amount of buffer

space needed at the receiving side in order to restore the original data transmission pattern.

### **ServiceType**

Contains the value of the level of service to negotiate for the flow. The following ServiceTypes have been defined.

#### **SERVICETYPE\_NOTRAFFIC**

In either the sending or receiving flow specification, this value indicates that there will be no traffic in this direction. On duplex capable media, this signals underlying software to set up unidirectional connections only.

#### **SERVICETYPE\_BESTEFFORT**

The service provider takes the **FLOWSPEC** as a guideline and makes reasonable efforts to maintain the level of service requested, without making any guarantees on packet delivery.

#### **SERVICETYPE\_CONTROLLEDLOAD**

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

#### **SERVICETYPE\_GUARANTEED**

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

#### **SERVICETYPE\_NETWORK\_UNAVAILABLE**

In either sending or receiving flow specifications, this value may be used by a service provider to indicate a loss of service in the corresponding direction.

#### **SERVICETYPE\_GENERAL\_INFORMATION**

All service types are supported for this traffic flow.

#### **SERVICETYPE\_NOCHANGE**

In either sending or receiving flow specifications, this level of service requests that the quality of service in the corresponding direction is not changed. ServiceType\_NoChange can be used when requesting a QOS change in one direction only, or when requesting a change only in the ProviderSpecific part of a QOS specification and not in the SendingFlowspec or the ReceivingFlowspec.

#### **SERVICETYPE\_IMMEDIATE\_TRAFFIC\_CONTROL**

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

### **MaxSduSize**

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

### **MinimumPolicedSize**

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

# 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

Windows Sockets allocates this structure is allocated. 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.

## IN\_ADDR

```
struct in_addr {
    union {
        struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
        struct { u_short s_w1,s_w2; } S_un_w;
        u_long S_addr;
    } S_un;
};
```

## LINGER

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

## Members

### **l\_onoff**

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.

## NS\_SERVICE\_INFO

The **NS\_SERVICE\_INFO** structure contains information about a network service or a network service type in the context of a specified name space, or a set of default name spaces.

```
typedef struct _NS_SERVICE_INFO {
    DWORD    dwNameSpace;
    SERVICE_INFO ServiceInfo;
} NS_SERVICE_INFO;
```

### Members

#### dwNameSpace

Specifies the name space or a set of default name spaces to which this service information applies.

Use one of the following constant values to specify a name space:

Value	Name Space
NS_DEFAULT	A set of default name spaces. 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_DNS	The Domain Name System used in the Internet to resolve the name of the host.
NS_MS	
NS_NDS	The NetWare 4 provider.
NS_NETBT	The NetBIOS over TCP/IP layer. The operating system registers their computer names with NetBIOS. This name space is used to convert a computer name to an IP address that uses this registration.
NS_NIS	
NS_SAP	The NetWare Service Advertising Protocol. This can access the Netware bindery, if appropriate. <code>NS_SAP</code> is a dynamic name space that enables the registration of services.
NS_STDA	

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.
NS_WINS	
NS_X500	

### ServiceInfo

A [SERVICE\\_INFO](#) structure that contains information about a network service or network service type.

### QuickInfo

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in nspapi.h.

### See Also

[SERVICE\\_INFO](#)

## PROTOCOL\_INFO

The **PROTOCOL\_INFO** structure contains information about a protocol.

```
typedef struct _PROTOCOL_INFO {
    DWORD dwServiceFlags;
    INT iAddressFamily;
    INT iMaxSockAddr;
    INT iMinSockAddr;
    INT iSocketType;
    INT iProtocol;
    DWORD dwMessageSize;
    LPTSTR lpProtocol;
} PROTOCOL_INFO;
```

### Members

#### dwServiceFlags

A set of bit flags that specify the services provided by the protocol. One or more of the following bit flags may be set:

<b>Value</b>	<b>Meaning</b>
XP_CONNECTIONLESS	If this flag is set, the protocol provides connectionless (datagram) service. If this flag is clear, the protocol provides connection-oriented data transfer.
XP_GUARANTEED_DELIVERY	If this flag is set, the protocol guarantees that all data sent will reach the intended destination. If this flag is clear, there is no such guarantee.
XP_GUARANTEED_ORDER	If this flag is set, the protocol guarantees that data will arrive in the order in which it was sent. Note that this characteristic does not guarantee delivery of the data, but guarantees only its order. If this flag is clear, the order of data sent is not guaranteed.
XP_MESSAGE_ORIENTED	If this flag is set, the protocol is message-oriented. A message-oriented protocol honors message boundaries. If this flag is clear, the protocol is stream-oriented, and the concept of message boundaries is irrelevant.
XP_PSEUDO_STREAM	If this flag is set, the protocol is a message-oriented protocol that ignores message boundaries for all receive operations.  This optional capability is useful when you do not want the protocol to frame messages. An application that requires stream-oriented characteristics can open a socket with type SOCK_STREAM for transport protocols that support this functionality, regardless of the value of <b>iSocketType</b> .
XP_GRACEFUL_CLOSE	If this flag is set, the protocol supports two-phase close operations, also known as "graceful" close operations. If this flag is clear, the protocol supports only abortive close operations.
XP_EXPEDITED_DATA	If this flag is set, the protocol supports expedited data, also known as "urgent data."
XP_CONNECT_DATA	If this flag is set, the protocol supports connect data.
XP_DISCONNECT_DATA	If this flag is set, the protocol supports disconnect data.
XP_SUPPORTS_BROADCAST	If this flag is set, the protocol supports a broadcast mechanism.
XP_SUPPORTS_MULTICAST	If this flag is set, the protocol supports a multicast mechanism.
XP_BANDWIDTH_ALLOCATION	If this flag is set, the protocol supports a mechanism for allocating a guaranteed bandwidth to an application.

XP_FRAGMENTATION	If this flag is set, the protocol supports message fragmentation; physical network MTU is hidden from applications.
XP_ENCRYPTS	If this flag is set, the protocol supports data encryption.

**iAddressFamily**

Specifies the value to pass as the *af* parameter when you call the **socket** function to open a socket for the protocol. This address family value uniquely defines the structure of Protocol addresses, also known as SOCKADDRs, used by the protocol.

**iMaxSockAddr**

Specifies the maximum length of a socket address supported by the protocol.

**iMinSockAddr**

Specifies the minimum length of a socket address supported by the protocol.

**iSocketType**

Specifies the value to pass as the *type* parameter when you call the **socket** function to open a socket for the protocol.

Note that if XP\_PSEUDO\_STREAM is set in **dwServiceFlags**, the application can specify SOCK\_STREAM as the *type* parameter to **socket**, regardless of the value of **iSocketType**.

**iProtocol**

Specifies the value to pass as the *protocol* parameter when you call the **socket** function to open a socket for the protocol.

**dwMessageSize**

Specifies the maximum message size supported by the protocol. This is the maximum size of a message that can be sent from or received by the host. For protocols that do not support message framing, the actual maximum size of a message that can be sent to a given address may be less than this value.

The following special message size values are defined:

Value	Meaning
0	The protocol is stream-oriented; the concept of message size is not relevant.
0xFFFFFFFF	The protocol is message-oriented, but there is no maximum message size.

**lpProtocol**

Points to a zero-terminated string that supplies a name for the protocol; for example, "SPX2."

**QuickInfo**

**Windows NT:** Yes

**Windows:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in nspapi.h.

**See Also**

[EnumProtocols](#), [socket](#)

## PROTOENT

```
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 returned pointer points to a structure that 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.

## 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;
```

## SERVENT

```
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.

## SERVICE\_ADDRESS

The **SERVICE\_ADDRESS** structure contains address information for a service. The structure can accommodate many types of interprocess communications (IPC) mechanisms and their address forms, including remote procedure calls (RPCs), named pipes, and sockets.

```
typedef struct _SERVICE_ADDRESS {
    DWORD    dwAddressType;
    DWORD    dwAddressFlags;
    DWORD    dwAddressLength;
    DWORD    dwPrincipalLength;
    BYTE     *lpAddress;
    BYTE     *lpPrincipal;
} SERVICE_ADDRESS;
```

### Members

#### dwAddressType

Specifies the address family that the socket address pointed to by **lpAddress** belongs to.

#### dwAddressFlags

A set of bit flags that specify properties of the address. The following bit flags are defined:

Value	Meaning
SERVICE_ADDRESS_FLAG_RPC_CN	If this bit flag is set, the service supports connection-oriented RPC over this transport protocol.
SERVICE_ADDRESS_FLAG_RPC_DG	If this bit flag is set, the service supports datagram-oriented RPC over this transport protocol.
SERVICE_ADDRESS_FLAG_RPC_NB	If this bit flag is set, the service supports NetBIOS RPC over this transport protocol.

#### dwAddressLength

Specifies the size, in bytes, of the address.

#### dwPrincipalLength

This member is reserved for future use. It must be zero.

#### lpAddress

Points to a socket address of the appropriate type.

#### lpPrincipal

This member is reserved for future use. It must be NULL.

## QuickInfo

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in nspapi.h.

## See Also

[SERVICE\\_ADDRESSES](#), [SERVICE\\_INFO](#)

# SERVICE\_ADDRESSES

The **SERVICE\_ADDRESSES** structure contains an array of [SERVICE\\_ADDRESS](#) data structures.

```
typedef struct _SERVICE_ADDRESSES {
    DWORD    dwAddressCount;
    SERVICE_ADDRESS  Addresses[1];
} SERVICE_ADDRESSES;
```

## Members

### dwAddressCount

Specifies the number of **SERVICE\_ADDRESS** structures in the **Addresses** array.

### Addresses

An array of [SERVICE\\_ADDRESS](#) data structures. Each **SERVICE\_ADDRESS** structure contains information about a network service address.

## QuickInfo

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in nspapi.h.

## See Also

[SERVICE\\_ADDRESS](#), [SERVICE\\_INFO](#)

# SERVICE\_INFO

The **SERVICE\_INFO** structure contains information about a network service or a network service type.

```
typedef struct _SERVICE_INFO {
    LPGUID    lpServiceType;
    LPTSTR    lpServiceName;
```

```

LPTSTR    lpComment;
LPTSTR    lpLocale;
DWORD     dwDisplayHint;
DWORD     dwVersion;
DWORD     dwTime;
LPTSTR    lpMachineName;
LPSERVICE_ADDRESSES lpServiceAddress;
BLOB      ServiceSpecificInfo;
} SERVICE_INFO;

```

## Members

### lpServiceType

Points to a GUID that is the type of the network service.

### lpServiceName

Points to a zero-terminated string that is the name of the network service.

If you are calling the **SetService** function with the **dwNameSpace** parameter set to **NS\_DEFAULT**, the network service name must be a common name. A common name is what the network service is commonly known as. An example of a common name for a network service is "My SQL Server".

If you are calling the **SetService** function with the **dwNameSpace** parameter set to a specific service name, the network service name can be a common name or a distinguished name. A distinguished name distinguishes the service to a unique location with a directory service. An example of a distinguished name for a network service is "MS\\SYS\\NT\\DEV\\My SQL Server".

### lpComment

Points to a zero-terminated string that is a comment or description for the network service. For example, "Used for development upgrades."

### lpLocale

Points to a zero-terminated string that contains locale information.

### dwDisplayHint

Specifies a hint as to how to display the network service in a network browsing user interface. This can be one of the following values:

Value	Meaning
RESOURCE_DISPLAYTYPE_DOMAIN	Display the network service as a domain.
RESOURCE_DISPLAYTYPE_FILE	Display the network service as a file.
RESOURCE_DISPLAYTYPE_GENERIC	The method used to display the object does not matter.
RESOURCE_DISPLAYTYPE_GROUP	Display the network service as a group.
RESOURCE_DISPLAYTYPE_SERVER	Display the network service as a server.
RESOURCE_DISPLAYTYPE_SHARE	Display the network service as a sharepoint.
RESOURCE_DISPLAYTYPE_TREE	Display the network service as a tree.

### dwVersion

Specifies version information for the network service. The high word of this value specifies

a major version number. The low word of this value specifies a minor version number.

#### **dwTime**

This member is reserved for future use. It must be set to zero.

#### **lpMachineName**

Points to a zero-terminated string that is the name of the computer on which the network service is running.

#### **lpServiceAddress**

Points to a [SERVICE\\_ADDRESSES](#) structure that contains an array of [SERVICE\\_ADDRESS](#) structures. Each [SERVICE\\_ADDRESS](#) structure contains information about a network service address.

A network service can call the [getsockname](#) function to determine the local address of the system.

#### **ServiceSpecificInfo**

A [BLOB](#) structure that specifies service-defined information.

Note that, in general, the data pointed to by the [BLOB](#) structure's [pBlobData](#) member must not contain any pointers. That is because only the network service knows the format of the data; copying the data without such knowledge would lead to pointer invalidation. If the data pointed to by [pBlobData](#) contains variable-sized elements, offsets from [pBlobData](#) can be used to indicate the location of those elements. There is one exception to this general rule: when [pBlobData](#) points to a [SERVICE\\_TYPE\\_INFO\\_ABS](#) structure. This is possible because both the [SERVICE\\_TYPE\\_INFO\\_ABS](#) structure, and any [SERVICE\\_TYPE\\_VALUE\\_ABS](#) structures it contains are predefined, and thus their formats are known to the operating system.

#### **QuickInfo**

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in nspapi.h.

#### **See Also**

[BLOB](#), [GetService](#), [NS\\_SERVICE\\_INFO](#), [SetService](#), [SERVICE\\_ADDRESS](#), [SERVICE\\_ADDRESSES](#), [SERVICE\\_TYPE\\_INFO\\_ABS](#), [SERVICE\\_TYPE\\_VALUE\\_ABS](#)

## **SERVICE\_TYPE\_INFO\_ABS**

The [SERVICE\\_TYPE\\_INFO\\_ABS](#) structure contains information about a network service type. You use a [SERVICE\\_TYPE\\_INFO\\_ABS](#) structure to add a network service type to a name space.

```
typedef struct _SERVICE_TYPE_INFO_ABS {
    LPCTSTR          lpTypeName;
    DWORD           dwValueCount;
    SERVICE_TYPE_VALUE_ABS Values[1];
} SERVICE_TYPE_INFO_ABS
```

#### **Members**

**lpTypeName**

Points to a zero-terminated string that is the name of the network service type. This name is the same in all name spaces, and is used by the **GetTypeByName** and **GetNameByType** functions.

**dwValueCount**

Specifies the number of [SERVICE\\_TYPE\\_VALUE\\_ABS](#) structures in the **Values** member array that follows **dwValueCount**.

**Values[1]**

An array of **SERVICE\_TYPE\_VALUE\_ABS** structures.

Each of these structures contains information about a service type value that the operating system or network service may need when an instance of this network service type is registered with a name space.

The information in these structures may be specific to a name-space. For example, if a network service uses the SAP name space, but does not have a **GUID** that contains the SAP identifier (SAPID), it defines the SAPID in a **SERVICE\_TYPE\_VALUE\_ABS** structure.

**Remarks**

When you use the **SetService** function to add a network service type to a name space, the **SERVICE\_TYPE\_INFO\_ABS** structure is passed as the **ServiceSpecificInfo BLOB** member of a **SERVICE\_INFO** structure. Although the **ServiceSpecificInfo** member generally should not contain pointers, an exception is made in the case of the **SERVICE\_TYPE\_INFO\_ABS** and **SERVICE\_TYPE\_VALUE\_ABS** structures.

**QuickInfo**

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in nspapi.h.

**See Also**

[SetService](#), [SERVICE\\_INFO](#), [SERVICE\\_TYPE\\_VALUE\\_ABS](#)

## SERVICE\_TYPE\_VALUE\_ABS

The **SERVICE\_TYPE\_VALUE\_ABS** structure contains information about a network-service type value. This information may be specific to a name space.

```
typedef struct _SERVICE_TYPE_VALUE_ABS {
    DWORD    dwNameSpace;
    DWORD    dwValueType;
    DWORD    dwValueSize;
    LPTSTR   lpValueName;
    PVOID    lpValue;
} SERVICE_TYPE_VALUE_ABS
```

## Members

### dwNameSpace

Specifies the name space, or a set of default name spaces, for which the network service type value is intended. Name-space providers will look only at values intended for their name space.

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 <i>dwNameSpace</i> .
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.

### dwValueType

Specifies the type of the value data. You can specify one of the following types:

Value	Meaning
REG_BINARY	Binary data in any form.
REG_DWORD	A 32-bit number.
REG_MULTI_SZ	An array of null-terminated strings, terminated by two null characters.
REG_SZ	A null-terminated string.

### dwValueSize

Specifies the size, in bytes, of the value data. In the case of REG\_SZ and REG\_MULTI\_SZ string data, the terminating characters are counted as part of the size.

### lpValueName

Points to a zero-terminated string that is the name of the value. This name is specific to a name space.

Several commonly used value name strings are associated with defined constants. These name strings include the following:

<b>Constant</b>	<b>Name String</b>
SERVICE_TYPE_VALUE_SAPID	"SapId"
SERVICE_TYPE_VALUE_CONN	"ConnectionOriented"
SERVICE_TYPE_VALUE_TCPPORT	"TcpPort"
SERVICE_TYPE_VALUE_UDPPORT	"UdpPort"

### **lpValue**

Points to the value data.

### **Remarks**

When you use the **SetService** function to add a network service type to a name space, a **SERVICE\_TYPE\_INFO\_ABS** structure is passed as the **ServiceSpecificInfo BLOB** member of a **SERVICE\_INFO** structure. Although the **ServiceSpecificInfo** member generally should not contain pointers, an exception is made in the case of the **SERVICE\_TYPE\_INFO\_ABS** and **SERVICE\_TYPE\_VALUE\_ABS** structures.

### **QuickInfo**

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in nspapi.h.

### **See Also**

[SetService](#), [SERVICE\\_INFO](#), [SERVICE\\_TYPE\\_INFO\\_ABS](#)

## **SOCKADDR**

The **SOCKADDR** structure varies depending on the protocol selected. Except for the `sa_family` field, **SOCKADDR** contents are expressed in network byte order.

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

In Windows Sockets 2, the **name** parameter is not strictly interpreted as a pointer to a **SOCKADDR** structure. It is presented in this manner for Windows Sockets compatibility. The actual structure is interpreted differently in the context of different address families. The only requirements are that the first `u_short` is the address family and the total size of the memory buffer in bytes is **namelen**.

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 ;
```

## TIMEVAL

```
struct timeval {
    long    tv_sec;           /* seconds */
    long    tv_usec;        /* and microseconds */
};
```

This structure is used in the **select** function. It is taken from the BSD file `sys/time.h`.

## TRANSMIT\_FILE\_BUFFERS

The **TRANSMIT\_FILE\_BUFFERS** structure specifies data to be transmitted before and after file data during a [TransmitFile](#) function file transfer operation.

```
typedef struct _TRANSMIT_FILE_BUFFERS {
    PVOID Head;
    DWORD HeadLength;
    PVOID Tail;
    DWORD TailLength;
} TRANSMIT_FILE_BUFFERS;
```

### Members

#### Head

Pointer to a buffer that contains data to be transmitted before the file data is transmitted.

#### HeadLength

Specifies the number of bytes of data in the buffer pointed to by the **Head** member that are to be transmitted.

#### Tail

Pointer to a buffer that contains data to be transmitted after the file data is transmitted.

## TailLength

Specifies the number of bytes of data in the buffer pointed to by the **Tail** member that are to be transmitted.

## QuickInfo

**Windows NT:** Yes

**Windows CE:** Unsupported.

**Header:** Declared in winsock.h.

## See Also

## [TransmitFile](#)

# WSABUF

```
typedef struct __WSABUF {
    u_longlen; // buffer length
    char FAR *buf; // pointer to buffer
} WSABUF, FAR * LPWSABUF;
```

## Members

### len

The length of the buffer.

### buf

A pointer to the buffer.

# 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;
```

The members of this structure are:

## Parameters

### wVersion

The version of the Windows Sockets specification that the WS2\_32.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*.

#### szDescription

A null-terminated ASCII string into which the WS2\_32.DLL copies a description of the Windows Sockets implementation. The text (up to 256 characters in length) can 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 WS2\_32.DLL copies relevant status or configuration information. The WS2\_32.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** 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).

## WSAEcomparator

```
typedef enum _WSAEcomparator {
    COMP_EQUAL = 0,
    COMP_NOTLESS
} WSAECOMPARATOR, *PWSAECOMPARATOR, *LPWSAECOMPARATOR;
```

## 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.

# WSANETWORKEVENTS

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

## Parameters

### *lNetworkEvents*

A value that indicates which of the FD\_XXX network events have occurred.

### *iErrorCodes*

An array that contains any associated error codes, with an array index that corresponds to the position of event bits in *lNetworkEvents*. The identifiers FD\_READ\_BIT, FD\_WRITE\_BIT and other can be used to index the *iErrorCodes* array.

# 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, FAR * 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.

# 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\_INTERRUPT**

This bit is reserved.

**XP1\_SUPPORT\_BROADCAST**

The protocol supports a broadcast mechanism.

**XP1\_SUPPORT\_MULTIPPOINT**

The protocol supports a multipoint or multicast mechanism. Control and data plane attributes are indicated below.

**XP1\_MULTIPPOINT\_CONTROL\_PLANE**

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

**XP1\_MULTIPPOINT\_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\_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\_INFOW** structures 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\_PROTO\_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\_INFOW** structure.

**[WSAPROTOCOLCHAIN](#)** ProtocolChain;

If the length of the chain is 0, this **WSAPROTOCOL\_INFOW** 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\_INFOW** 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. There is no standard provision to determine the maximum inbound message size. The following special values are defined:

0

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

0x1

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

0xFFFFFFFF

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

**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;

```

## 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;

```

### Parameters

#### *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 can 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*<sup>1</sup>

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, can 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 that 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.

## WSAServiceClassInfo

```
typedef struct _WSAServiceClassInfo {
    LPGUID          lpServiceClassId;
    LPWSTR          lpzServiceClassName;
    DWORD           dwCount;
    LPWSANSCLASSINFO lpClassInfos;
} WSA_SERVICE_CLASS_INFO, *PWSA_SERVICE_CLASS_INFO, *LPWSA_SERVICE_CLASS_INFO;
```

## WSATHREADID

```
typedef struct _WSATHREADID {
    HANDLE ThreadHandle;
    DWORD Reserved;
} WSATHREADID, FAR * LPWSATHREADID;
```