

# Contents

## Windows Sockets 2

[What's New for Windows Sockets](#)

[Winsock Network Protocol Support in Windows](#)

[About Winsock](#)

[Winsock Programming Considerations](#)

[Windows Sockets 2 Architecture](#)

[Socket Handles](#)

[Scatter/Gather I/O](#)

[Protocol-Independent Multicast and Multipoint](#)

[Flow Specification Quality of Service](#)

[Provider-Specific Extension Mechanism](#)

[Shared Sockets](#)

[Connection Setup and Teardown](#)

[Graceful Shutdown, Linger Options, and Socket Closure](#)

[Protocol-Independent Out-of-Band Data](#)

[Debug and Trace Facilities](#)

[Windows Sockets Compatibility Issues](#)

[Default State for a Socket's Overlapped Attribute](#)

[Windows Sockets 1.1 Blocking Routines and EINPROGRESS](#)

[Handling Winsock Errors](#)

[Simultaneous Access to Multiple Transport Protocols](#)

[Layered Protocols and Protocol Chains](#)

[Using Multiple Protocols](#)

[Multiple Provider Restrictions on Select](#)

[Overlapped I/O and Event Objects](#)

[Event Objects](#)

[Receiving Completion Indications](#)

[Asynchronous Notification Using Event Objects](#)

[Registration and Name Resolution](#)

## Protocol-Independent Name Resolution

Name Resolution Model

Summary of Name Resolution Functions

Name Resolution Data Structures

## Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 API

Basic Approach for GetXbyY in the API

getprotobynumber and getprotobyname Functions in the API

getservbyname and getservbyport Functions in the API

gethostbyname Function in the API

gethostbyaddr Function in the API

gethostname Function in the API

## Multipoint and Multicast Semantics

Multipoint Taxonomy

Windows Sockets 2 Interface Elements for Multipoint and Multicast

Attributes in WSAPROTOCOL\_INFO Structure

Flag Bits for WSASocket

SIO\_MULTIPOINT\_LOOPBACK Command Code for WSAIoctl

SIO\_MULTICAST\_SCOPE Command Code for WSAIoctl

Semantics for Joining Multipoint Leaves

Using WSAJoinLeaf

Semantic differences between multipoint sockets and regular sockets

How multipoint protocols support these extensions

IP Multicast

ATM Point to Multipoint

## Internet Protocol Version 6 (IPv6)

Using IPv6

Netsh.exe

Net.exe

Ipv6.exe

Ipsec6.exe

Using Internet Explorer to Access IPv6 Websites

Recommended Configurations

[Configuration 1: Single Subnet with Link-local Addresses](#)

[Configuration 2: IPv6 Traffic Between Nodes on Different Subnets of an IPv4 Internetwork \(6to4\)](#)

[Configuration 3: Using IPsec Between Two Local-link Hosts](#)

[Additional IPv6 Topics](#)

[IPv6 Router Advertisements](#)

[IPv6 Site Prefixes](#)

[IPv6 Link-local and Site-local Addresses](#)

[IPv6 Multicast Destination Addresses](#)

[IPv6 Automatic Tunneling and 6to4](#)

[IPv6 Forwarding Tunneled Packets](#)

[Network Location Awareness Service Provider \(NLA\)](#)

[The Role of NLA](#)

[Querying NLA](#)

[Notification from NLA](#)

[Registering a Service Instance with NLA](#)

[About the Winsock SPI](#)

[Function Interface Model](#)

[Service Provider Ordering](#)

[Transport Service Providers](#)

[Transport Division of Responsibilities Between DLL and Service Providers](#)

[Transport Mapping Between API and SPI Functions](#)

[Function Extension Mechanism in the SPI](#)

[Transport Configuration and Installation](#)

[Winsock Transport Service Provider Requirements](#)

[Service Provider Activation](#)

[Socket Creation and Descriptor Management](#)

[Blocking Operations](#)

[Event Objects in the Windows Sockets 2 SPI](#)

[Notification of Network Events](#)

[Socket Groups in the Windows Sockets 2 SPI](#)

[Quality of Service in the Windows Sockets 2 SPI](#)

[Socket Connections on Connection-Oriented Protocols](#)

[Socket Connections on Connectionless Protocols](#)

[Socket I/O](#)

[Shared Sockets in the SPI](#)

[Protocol-Independent Multicast and Multipoint in the SPI](#)

[Socket Options and IOCTLs](#)

[Summary of SPI Functions](#)

[Namespace Service Providers](#)

[Name Resolution Model for the SPI](#)

[Types of Namespaces in the SPI](#)

[Namespace Organization in the SPI](#)

[Namespace Provider Architecture in the SPI](#)

[Name resolution for DLL and service providers](#)

[Name Resolution Mapping Between API and SPI Functions](#)

[Name Resolution Configuration and Installation](#)

[Winsock Namespace Service Provider Requirements](#)

[Summary of Namespace Provider Functions](#)

[Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 SPI](#)

[Sample Code for a Service Provider](#)

[Using Winsock](#)

[Getting Started With Winsock](#)

[About Servers and Clients](#)

[Creating a Basic Winsock Application](#)

[Initializing Winsock](#)

[Winsock Client Application](#)

[Creating a Socket for the Client](#)

[Connecting to a Socket](#)

[Sending and Receiving Data on the Client](#)

[Disconnecting the Client](#)

[Winsock Server Application](#)

[Creating a Socket for the Server](#)

[Binding a Socket](#)

[Listening on a Socket](#)

- [Accepting a Connection](#)
- [Receiving and Sending Data on the Server](#)
- [Disconnecting the Server](#)
- [Running the Winsock Client and Server Code Sample](#)
  - [Complete Winsock Client Code](#)
  - [Complete Winsock Server Code](#)
- [Secure Winsock Programming](#)
  - [Using SO\\_REUSEADDR and SO\\_EXCLUSIVEADDRUSE](#)
  - [Winsock Secure Socket Extensions](#)
    - [Using Secure Socket Extensions](#)
    - [Advanced Winsock Samples Using Secure Socket Extensions](#)
- [Porting Socket Applications to Winsock](#)
  - [Socket Data Type](#)
  - [Select and FD\\_\\*](#)
  - [Error Codes - errno, h\\_errno and WSAGetLastError](#)
  - [Pointers](#)
  - [Renamed Functions](#)
  - [Maximum Number of Sockets Supported](#)
  - [Include Files](#)
  - [Return Values on Function Failure](#)
  - [Raw Sockets](#)
  - [Byte Ordering](#)
  - [Extended Byte-Order Conversion Routines](#)
- [IPv6 Guide for Windows Sockets Applications](#)
  - [Changing Data Structures for IPv6 Winsock Applications](#)
  - [Function Calls for IPv6 Winsock Applications](#)
  - [Use of Hardcoded IPv4 Addresses](#)
  - [User Interface Issues for IPv6 Winsock Applications](#)
  - [Underlying Protocols for IPv6 Winsock Applications](#)
  - [Dual-Stack Sockets for IPv6 Winsock Applications](#)
  - [Using the Checkv4.exe Utility](#)
- [Appendix A: IPv4-only Source Code](#)

- [IPv4-only Client Code](#)
- [IPv4-only Server Code](#)
- [Appendix B: IP-version Agnostic Source Code](#)
  - [IPv6-Enabled Client Code](#)
  - [IPv6-Enabled Server Code](#)
- [High-performance Windows Sockets Applications](#)
  - [Network Terminology](#)
  - [Performance Dimensions](#)
    - [Performance Needs: Users and Administrators](#)
    - [Transactional versus Streaming Applications](#)
    - [Different Network Environments](#)
  - [TCP/IP Characteristics](#)
  - [Application Behavior](#)
    - [TCP/IP-specific Issues](#)
    - [Recognizing Slow Applications](#)
    - [Calculating Overhead with Netstat](#)
  - [Improving a Slow Application](#)
    - [The Baseline Version: A Very Poor Performing Application](#)
    - [Revision One: Cleaning up the Obvious](#)
    - [Revision Two: Redesigning for Fewer Connects](#)
    - [Revision Three: Compressed Block Send](#)
    - [Future Improvements](#)
  - [Best Practices for Interactive Applications](#)
  - [Categorizing layered service providers and apps](#)
  - [Multicast Programming](#)
    - [MLD and IGMP Using Windows Sockets](#)
    - [Multicast Socket Option Behavior](#)
    - [Multicast Programming Sample](#)
    - [Final-State-Based Multicast Programming](#)
    - [Porting Broadcast Applications to IPv6](#)
    - [Reliable Multicast Programming \(PGM\)](#)
    - [PGM Senders and Receivers](#)

[PGM Sender Options](#)

[Sending and Receiving PGM Data](#)

[Multihoming and PGM](#)

[PGM Socket Options](#)

[Winsock Tracing](#)

[Winsock Tracing Levels](#)

[Control of Winsock Tracing](#)

[Winsock Network Event Tracing Details](#)

[Winsock Catalog Change Tracing Details](#)

[Winsock timestamping](#)

[Winsock explicit congestion notification](#)

[Winsock socket state notifications](#)

[Winsock Reference](#)

[Socket Options](#)

[IPPROTO\\_IP Socket Options](#)

[IP\\_PKTINFO](#)

[IPPROTO\\_IPV6 Socket Options](#)

[IPV6\\_PKTINFO](#)

[IPV6\\_PROTECTION\\_LEVEL](#)

[IPPROTO\\_RM Socket Options](#)

[IPPROTO\\_TCP Socket Options](#)

[IPPROTO\\_UDP Socket Options](#)

[NSPROTO\\_IPX Socket Options](#)

[SOL\\_APPLETALK Socket Options](#)

[SOL\\_IRLMP Socket Options](#)

[SOL\\_SOCKET Socket Options](#)

[SO\\_BSP\\_STATE](#)

[SO\\_CONDITIONAL\\_ACCEPT](#)

[SO\\_EXCLUSIVEADDRUSE](#)

[SO\\_KEEPALIVE](#)

[SO\\_PORT\\_SCALABILITY](#)

[IP\\_DSCP\\_TRAFFIC\\_TYPE](#)

## Winsock IOCTLs

SIO\_ACQUIRE\_PORT\_RESERVATION  
SIO\_ADDRESS\_LIST\_QUERY  
SIO\_APPLY\_TRANSPORT\_SETTING  
SIO\_ASSOCIATE\_PORT\_RESERVATION  
SIO\_IDEAL\_SEND\_BACKLOG\_CHANGE  
SIO\_IDEAL\_SEND\_BACKLOG\_QUERY  
SIO\_KEEPALIVE\_VALS  
SIO\_LOOPBACK\_FAST\_PATH  
SIO\_QUERY\_RSS\_PROCESSOR\_INFO  
SIO\_QUERY\_TRANSPORT\_SETTING  
SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_RECORDS  
SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_CONTEXT  
SIO\_RECVALL  
SIO\_RELEASE\_PORT\_RESERVATION  
SIO\_SET\_COMPATIBILITY\_MODE  
SIO\_SET\_WFP\_CONNECTION\_REDIRECT\_RECORDS  
SIO\_TCP\_INFO  
SIO\_TCP\_INITIAL\_RTO

## Winsock Annexes

### Winsock ATM Annex

Winsock ATM Controls  
Winsock ATM Function Details  
Winsock ATM QoS Extension  
AAL Parameters  
ATM Traffic Descriptor  
Broadband Bearer Capability  
Broadband High Layer Information  
Broadband Lower Layer Information  
Called Party Number  
Called Party Subaddress  
Calling Party Number

[Calling Party Subaddress](#)

[Quality of Service Parameter](#)

[Transit Network Selection](#)

[Cause](#)

[Winsock ATM Structures](#)

[ATM\\_ADDRESS](#)

[ATM\\_BHLI](#)

[ATM\\_BLLI](#)

[sockaddr\\_atm](#)

[Winsock IPX/SPX Annex](#)

[AF\\_IPX Address Family](#)

[IPX Family of Protocol Identifiers](#)

[Broadcast to Local Network](#)

[All Routes Broadcast](#)

[Directed Broadcast](#)

[About Media Packet Size](#)

[How Packet Size Affects Protocols](#)

[Winsock IPX/SPX Structures](#)

[IPX\\_ADDRESS\\_DATA](#)

[IPX\\_NETNUM\\_DATA](#)

[IPX\\_SPXCONNSTATUS\\_DATA](#)

[Winsock TCP/IP Annex](#)

[Using UNIX ioctl in Winsock](#)

[Winsock TCP/IP Function Details](#)

[Multicast](#)

[TCP/IP Raw Sockets](#)

[IPv6 Support](#)

[Text Representation of IPv6 Addresses](#)

[Winsock TCP/IP Structures](#)

[INTERFACE\\_INFO](#)

[INTERFACE\\_INFO\\_EX](#)

[sockaddr\\_gen](#)

## Winsock Enumerations

CONTROL\_CHANNEL\_TRIGGER\_STATUS  
eWINDOW\_ADVANCE\_METHOD  
GUARANTEE  
MULTICAST\_MODE\_TYPE  
NAPI\_PROVIDER\_LEVEL  
NAPI\_PROVIDER\_TYPE  
RIO\_NOTIFICATION\_COMPLETION\_TYPE  
SOCKET\_SECURITY\_PROTOCOL  
SOCKET\_USAGE\_TYPE  
TCPSTATE  
WSAECOMPARATOR  
WSC\_PROVIDER\_INFO\_TYPE

## Winsock Functions

accept  
AcceptEx  
bind  
closesocket  
connect  
LPFN\_CONNECTEX  
DisconnectEx  
EnumProtocols  
freeaddrinfo  
FreeAddrInfoEx  
FreeAddrInfoW  
gai\_strerror  
GetAcceptExSockaddrs  
GetAddressByName  
getaddrinfo  
GetAddrInfoEx  
GetAddrInfoExCancel  
GetAddrInfoExOverlappedResult

[GetAddrInfoW](#)  
[gethostbyaddr](#)  
[gethostbyname](#)  
[gethostname](#)  
[GetHostNameW](#)  
[GetNameByType](#)  
[getnameinfo](#)  
[getipv4sourcefilter](#)  
[GetNameInfoW](#)  
[getpeername](#)  
[getprotobynumber](#)  
[getservbyname](#)  
[getservbyport](#)  
[GetService](#)  
[getsockname](#)  
[getsockopt](#)  
[getsourcefilter](#)  
[GetTypeByName](#)  
[htond](#)  
[htonf](#)  
[htonl](#)  
[htonll](#)  
[htons](#)  
[inet\\_addr](#)  
[inet\\_ntoa](#)  
[InetNtop](#)  
[InetPton](#)  
[ioctlsocket](#)  
[listen](#)  
[ntohd](#)  
[ntohf](#)

ntohl  
ntohll  
 ntohs  
recv  
recvfrom  
RIOCloseCompletionQueue  
RIOCreateCompletionQueue  
RIOCreateRequestQueue  
RIODequeueCompletion  
RIODeregisterBuffer  
RIONotify  
RIOReceive  
RIOReceiveEx  
RIORegisterBuffer  
RIOResizeCompletionQueue  
RIOResizeRequestQueue  
RIOSend  
RIOSendEx  
select  
send  
sendto  
SetAddrInfoEx  
setipv4sourcefilter  
SetService  
SetSocketMediaStreamingMode  
setsockopt  
setsourcefilter  
shutdown  
socket  
TransmitFile  
LPFN\_TRANSMITPACKETS  
WSAAccept

WSAAddressToString  
WSAAAsyncGetHostByAddr  
WSAAAsyncGetHostByName  
WSAAAsyncGetProtoByName  
WSAAAsyncGetProtoByNumber  
WSAAAsyncGetServByName  
WSAAAsyncGetServByPort  
WSAAAsyncSelect  
WSACancelAsyncRequest  
WSACancelBlockingCall  
WSACleanup  
WSACloseEvent  
WSAConnect  
WSAConnectByList  
WSAConnectByName  
WSACreateEvent  
WSADeleteSocketPeerTargetName  
WSADuplicateSocket  
WSAEnumNameSpaceProviders  
WSAEnumNameSpaceProvidersEx  
WSAEnumNetworkEvents  
WSAEnumProtocols  
WSAEventSelect  
\_WSAFDIsSet  
WSAGetFailConnectOnIcmpError  
WSAGetIcmpErrorInfo  
WSAGetIPUserMtu  
WSAGetLastError  
WSAGetOverlappedResult  
WSAGetQOSByName  
WSAGetServiceClassInfo  
WSAGetServiceClassNameById

[WSAGetUdpRecvMaxCoalescedSize](#)  
[WSAGetUdpSendMessageSize](#)  
[WSAHtonl](#)  
[WSAHtons](#)  
[WSAImpersonateSocketPeer](#)  
[WSAInstallServiceClass](#)  
[WSAIoctl](#)  
[WSAIsBlocking](#)  
[WSAJoinLeaf](#)  
[WSALookupServiceBegin](#)  
[WSALookupServiceEnd](#)  
[WSALookupServiceNext](#)  
[WSANSPIoctl](#)  
[WSANtohl](#)  
[WSANtohs](#)  
[WSAPoll](#)  
[WSAQuerySocketSecurity](#)  
[WSAProviderConfigChange](#)  
[WSARecv](#)  
[WSARecvDisconnect](#)  
[WSARecvEx](#)  
[WSARecvFrom](#)  
[LPFN\\_WSARECVMSG \(WSARecvMsg\)](#)  
[WSARemoveServiceClass](#)  
[WSAResetEvent](#)  
[WSARevertImpersonation](#)  
[WSASend](#)  
[WSASendDisconnect](#)  
[WSASendMsg](#)  
[WSASendTo](#)  
[WSASetBlockingHook](#)  
[WSASetEvent](#)

[WSASetFailConnectOnIcmpError](#)

[WSASetIPUserMtu](#)

[WSASetLastError](#)

[WSASetService](#)

[WSASetSocketPeerTargetName](#)

[WSASetSocketSecurity](#)

[WSASetUdpRecvMaxCoalescedSize](#)

[WSASetUdpSendMessageSize](#)

[WSASocket](#)

[WSAStartup](#)

[WSAStringToAddress](#)

[WSAUUnhookBlockingHook](#)

[WSAWaitForMultipleEvents](#)

## Winsock Structures

[addrinfo](#)

[addrinfoW](#)

[addrinfoex](#)

[addrinfoex2](#)

[addrinfoex3](#)

[addrinfoex4](#)

[AFPROTOCOLS](#)

[ASSOCIATE\\_NAMERES\\_CONTEXT\\_INPUT](#)

[BLOB](#)

[CSADDR\\_INFO](#)

[fd\\_set](#)

[GROUP\\_FILTER](#)

[GROUP\\_REQ](#)

[GROUP\\_SOURCE\\_REQ](#)

[hostent](#)

[in\\_addr](#)

[in\\_pktinfo](#)

[in6\\_addr](#)

in6\_pktinfo  
INET\_PORT\_RANGE  
INET\_PORT\_RESERVATION\_INSTANCE  
INET\_PORT\_RESERVATION\_TOKEN  
ip\_mreq  
ip\_mreq\_source  
ip\_msfilter  
ipv6\_mreq  
linger  
NAPI\_DOMAIN\_DESCRIPTION\_BLOB  
NAPI\_PROVIDER\_INSTALLATION\_BLOB  
NS\_SERVICE\_INFO  
PROTOCOL\_INFO  
protoent  
REAL\_TIME\_NOTIFICATION\_SETTING\_INPUT  
REAL\_TIME\_NOTIFICATION\_SETTING\_OUTPUT  
RIO\_EXTENSION\_FUNCTION\_TABLE  
RIO\_BUF  
RIO\_BUFFERID  
RIO\_CQ  
RIO\_NOTIFICATION\_COMPLETION  
RIO\_RQ  
RIORESULT  
RM\_FEC\_INFO  
RM\_RECEIVER\_STATS  
RM\_SEND\_WINDOW  
RM\_SENDER\_STATS  
servent  
SERVICE\_ADDRESS  
SERVICE\_ADDRESSES  
SERVICE\_INFO  
SERVICE\_TYPE\_INFO\_ABS

SERVICE\_TYPE\_VALUE\_ABS  
sockaddr  
SOCKADDR\_IRDA  
SOCKADDR\_STORAGE  
SOCKET\_ADDRESS  
SOCKET\_ADDRESS\_LIST  
SOCKET\_PEER\_TARGET\_NAME  
SOCKET\_PROCESSOR\_AFFINITY  
SOCKET\_SECURITY\_QUERY\_INFO  
SOCKET\_SECURITY\_QUERY\_TEMPLATE  
SOCKET\_SECURITY\_SETTINGS  
SOCKET\_SECURITY\_SETTINGS\_IPSEC  
TCP\_INFO\_v0  
TCP\_INITIAL\_RTO\_PARAMETERS  
timeval  
TRANSMIT\_FILE\_BUFFERS  
TRANSMIT\_PACKETS\_ELEMENT  
TRANSPORT\_SETTING\_ID  
WSABUF  
WSACOMPLETION  
WSADATA  
WSAMSG  
WSANAMESPACE\_INFO  
WSANAMESPACE\_INFOEX  
WSANETWORKEVENTS  
WSANSCLASSINFO  
WSAOVERLAPPED  
WSAPOLLFD  
WSAPROTOCOL\_INFO  
WSAPROTOCOL\_INFOW  
WSAPROTOCOLCHAIN  
WSAQUERYSET

[WSAQUERYSET2](#)

[WSASERVICECLASSINFO](#)

[WSAVERSION](#)

[Winsock Tracing Events](#)

[AFD\\_EVENT\\_CREATE](#)

[AFD\\_EVENT\\_CLOSE](#)

[WINSOCK\\_WS2HELP\\_LSP\\_INSTALL](#)

[WINSOCK\\_WS2HELP\\_LSP\\_REMOVE](#)

[WINSOCK\\_WS2HELP\\_LSP\\_DISABLE](#)

[WINSOCK\\_WS2HELP\\_LSP\\_RESET](#)

[Winsock SPI](#)

[Winsock SPI Functions](#)

[LPNSPCLEANUP](#)

[LPNSPGETSERVICECLASSINFO](#)

[LPNSPIOCTL](#)

[LPNSPINSTALLSERVICECLASS](#)

[LPNSPLOOKUPSERVICEBEGIN](#)

[LPNSPLOOKUPSERVICEEND](#)

[LPNSPLOOKUPSERVICENEXT](#)

[LPNSPREMOVESERVICECLASS](#)

[LPNSPSETSERVICE](#)

[NSPStartup](#)

[LPNSPV2CLEANUP](#)

[LPNSPV2CLIENTSESSIONRUNDOWN](#)

[LPNSPV2LOOKUPSERVICEBEGIN](#)

[LPNSPV2LOOKUPSERVICEEND](#)

[LPNSPV2LOOKUPSERVICENEXTEX](#)

[LPNSPV2SETSERVICEEX](#)

[LPNSPV2STARTUP](#)

[WPUCloseEvent](#)

[WPUCloseSocketHandle](#)

[WPUCloseThread](#)

[WPUCCompleteOverlappedRequest](#)  
[WPUCreateEvent](#)  
[WPUCreateSocketHandle](#)  
[WPUFDIsSet](#)  
[WPUGetProviderPath](#)  
[WPUModifyIFSHandle](#)  
[WPUOpenCurrentThread](#)  
[WPUPostMessage](#)  
[WPUQueryBlockingCallback](#)  
[WPUQuerySocketHandleContext](#)  
[WPUQueueApc](#)  
[WPURestEvent](#)  
[WPUSetEvent](#)  
[WSAAdvertiseProvider](#)  
[WSAProviderCompleteAsyncCall](#)  
[WSAUnadvertiseProvider](#)  
[WSCDeinstallProvider](#)  
[WSCDeinstallProvider32](#)  
[WSCEnableNSProvider](#)  
[WSCEnableNSProvider32](#)  
[WSCEnumNameSpaceProviders32](#)  
[WSCEnumNameSpaceProvidersEx32](#)  
[WSCEnumProtocols](#)  
[WSCEnumProtocols32](#)  
[WSCGetApplicationCategory](#)  
[WSCGetProviderInfo](#)  
[WSCGetProviderInfo32](#)  
[WSCGetProviderPath](#)  
[WSCGetProviderPath32](#)  
[WSCInstallNameSpace](#)  
[WSCInstallNameSpace32](#)  
[WSCInstallNameSpaceEx](#)

WSCInstallNameSpaceEx32  
WSCInstallProvider  
WSCInstallProvider64\_32  
WSCInstallProviderAndChains  
WSCInstallProviderAndChains64\_32  
WSCInstallQOSTemplate  
WSCRemoveQOSTemplate  
WSCSetApplicationCategory  
WSCSetProviderInfo  
WSCSetProviderInfo32  
WSCUnInstallNameSpace  
WSCUnInstallNameSpace32  
WSCUpdateProvider  
WSCUpdateProvider32  
WSCWriteNameSpaceOrder  
WSCWriteNameSpaceOrder32  
WSCWriteProviderOrder  
WSCWriteProviderOrder32  
LPWSPACCEPT  
LPWSPADDRESSTOSTRING  
LPWSPASYNCSELECT  
LPWSPBIND  
LPWSPCANCELBLOCKINGCALL  
LPWSPCLEANUP  
LPWSPCLOSESOCKET  
LPWSPCONNECT  
LPWSPDUPLICATESOCKET  
LPWSPENUMNETWORKEVENTS  
LPWSPEVENTSELECT  
LPWSPGETOVERLAPPEDRESULT  
LPWSPGETPEERNAME  
LPWSPGETQOSBYNAME

LPWSPGETSOCKNAME  
LPWSPGETSOCKOPT  
LPWSPIOCTL  
LPWSPJOINLEAF  
LPWSPLISTEN  
LPWSPRECV  
LPWSPRECVDISCONNECT  
LPWSPRECVFROM  
LPWSPSELECT  
LPWSPSEND  
LPWSPSENDDDISCONNECT  
LPWSPSENDTO  
LPWSPSETSOCKOPT  
LPWSPSHUTDOWN  
LPWSPSOCKET  
WSPStartup  
LPWSPSTRINGTOADDRESS  
Winsock SPI Structures  
NSP\_ROUTINE  
NSPV2\_ROUTINE  
WSATHREADID  
WSC\_PROVIDER\_AUDIT\_INFO  
WSPDATA  
WSPPROC\_TABLE  
WSPUPCALLTABLE  
Windows Sockets Error Codes

# Windows Sockets 2

3/5/2021 • 2 minutes to read • [Edit Online](#)

## Purpose

Windows Sockets 2 (Winsock) enables programmers to create advanced Internet, intranet, and other network-capable applications to transmit application data across the wire, independent of the network protocol being used. With Winsock, programmers are provided access to advanced Microsoft® Windows® networking capabilities such as multicast and Quality of Service (QoS).

Winsock follows the Windows Open System Architecture (WOSA) model; it defines a standard service provider interface (SPI) between the application programming interface (API), with its exported functions and the protocol stacks. It uses the sockets paradigm that was first popularized by Berkeley Software Distribution (BSD) UNIX. It was later adapted for Windows in Windows Sockets 1.1, with which Windows Sockets 2 applications are backward compatible. Winsock programming previously centered around TCP/IP. Some programming practices that worked with TCP/IP do not work with every protocol. As a result, the Windows Sockets 2 API adds functions where necessary to handle several protocols.

## Developer audience

Windows Sockets 2 is designed for use by C/C++ programmers. Familiarity with Windows networking is required.

## Run-time requirements

Windows Sockets 2 can be used on all Windows platforms. Where certain implementations or capabilities of Windows Sockets 2 platform restrictions do exist, they are clearly noted in the documentation.

## In this section

TOPIC	DESCRIPTION
<a href="#">What's New for Windows Sockets</a>	Information on new features for Windows Sockets.
<a href="#">Winsock Network Protocol Support in Windows</a>	Information on network protocol support for Windows Sockets on different versions of Windows.
<a href="#">About Winsock</a>	General information on Windows Sockets programming considerations, architecture, and capabilities available to developers.
<a href="#">Using Winsock</a>	Procedures and programming techniques used with Windows Sockets. This section includes basic Winsock programming techniques, such as <a href="#">Getting Started With Winsock</a> , as well as advanced techniques useful for experienced Winsock developers.
<a href="#">Winsock Reference</a>	Documentation of the Windows Sockets API.

## Related topics

[IP Helper](#)

[Quality of Service](#)

# What's New for Windows Sockets

3/5/2021 • 6 minutes to read • [Edit Online](#)

## Updated for Windows 8.1 and Windows Server 2012 R2

The following functions are supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later. Microsoft Visual Studio 2013 Update 3 or later is required for Windows Store apps.

- [\\_\\_WSAFDIsSet](#)
- [accept](#)
- [AcceptEx](#)
- [bind](#)
- [closesocket](#)
- [connect](#)
- [ConnectEx](#)
- [DisconnectEx](#)
- [freeaddrinfo](#)
- [FreeAddrInfoExW](#)
- [freeaddrinfoW](#)
- [GetAcceptExSockaddrs](#)
- [getaddrinfo](#)
- [GetAddrInfoExCancel](#)
- [GetAddrInfoExOverlappedResult](#)
- [GetAddrInfoExW](#)
- [GetAddrInfoW](#)
- [gethostbyaddr](#)
- [gethostbyname](#)
- [gethostname](#)
- [GetHostNameW](#)
- [getipv4sourcefilter](#)
- [getnameinfo](#)
- [GetNameInfoW](#)
- [getpeername](#)
- [getprotobynumber](#)
- [getprotobyname](#)
- [getservbyname](#)
- [getservbyport](#)
- [getsockname](#)
- [getsockopt](#)
- [getsourcefilter](#)
- [htond](#)
- [htonf](#)
- [htonl](#)
- [htonll](#)
- [htons](#)

- [inet\\_addr](#)
- [inet\\_ntoa](#)
- [inet\\_ntop](#)
- [inet\\_pton](#)
- [InetNtopW](#)
- [InetPtonW](#)
- [ioctlsocket](#)
- [listen](#)
- [ntohd](#)
- [ntohf](#)
- [ntohl](#)
- [ntohll](#)
- [ntohs](#)
- [recv](#)
- [recvfrom](#)
- [RIOCloseCompletionQueue](#)
- [RIOCreateCompletionQueue](#)
- [RIOCreateRequestQueue](#)
- [RIODequeueCompletion](#)
- [RIODeregisterBuffer](#)
- [RIONotify](#)
- [RIOReceive](#)
- [RIOReceiveEx](#)
- [RIORegisterBuffer](#)
- [RIOResizeCompletionQueue](#)
- [RIOResizeRequestQueue](#)
- [RIOSend](#)
- [RIOSendEx](#)
- [select](#)
- [send](#)
- [sendto](#)
- [SetAddrInfoExW](#)
- [setipv4sourcefilter](#)
- [setsockopt](#)
- [setsourcefilter](#)
- [shutdown](#)
- [socket](#)
- [TransmitFile](#)
- [TransmitPackets](#)
- [WSAAccept](#)
- [WSAAddressToStringW](#)
- [WSACleanup](#)
- [WSACloseEvent](#)
- [WSAConnect](#)
- [WSAConnectByList](#)
- [WSAConnectByNameW](#)

- [WSACreateEvent](#)
- [WSADuplicateSocketW](#)
- [WSAEnumNameSpaceProvidersExW](#)
- [WSAEnumNameSpaceProvidersW](#)
- [WSAEnumNetworkEvents](#)
- [WSAEnumProtocolsW](#)
- [WSAEVENTSELECT](#)
- [WSAGetLastError](#)
- [WSAGetOverlappedResult](#)
- [WSAHtonl](#)
- [WSAHtons](#)
- [WSAloctl](#)
- [WSAJoinLeaf](#)
- [WSALookupServiceBeginW](#)
- [WSALookupServiceEnd](#)
- [WSALookupServiceNextW](#)
- [WSANSPIoctl](#)
- [WSANtohl](#)
- [WSANtohs](#)
- [WSAPoll](#)
- [WSAProviderConfigChange](#)
- [WSARecv](#)
- [WSARecvFrom](#)
- [LPFN\\_WSARECVMSG \(WSARecvMsg\)](#)
- [WSAResetEvent](#)
- [WSASend](#)
- [WSASendMsg](#)
- [WSASendTo](#)
- [WSASetEvent](#)
- [WSASetLastError](#)
- [WSASetServiceW](#)
- [WSASocketW](#)
- [WSAStartup](#)
- [WSAStringToAddressW](#)
- [WSAWaitForMultipleEvents](#)

## Updated for Windows Phone 8

The following functions are supported for Windows Phone Store apps on Windows Phone 8 and later.

- [\\_\\_WSAFDIsSet](#)
- [AcceptEx](#)
- [bind](#)
- [closesocket](#)
- [connect](#)
- [ConnectEx](#)
- [DisconnectEx](#)
- [freeaddrinfo](#)

- [getaddrinfo](#)
- [gethostbyaddr](#)
- [gethostbyname](#)
- [gethostname](#)
- [GetHostNameW](#)
- [getipv4sourcefilter](#)
- [getnameinfo](#)
- [GetNameInfoW](#)
- [getpeername](#)
- [getprotobynumber](#)
- [getservbyname](#)
- [getservbyport](#)
- [getsockname](#)
- [getsockopt](#)
- [getsourcefilter](#)
- [htond](#)
- [htonf](#)
- [htonll](#)
- [inet\\_addr](#)
- [inet\\_ntoa](#)
- [ioctlsocket](#)
- [listen](#)
- [ntohd](#)
- [ntohf](#)
- [ntohll](#)
- [recv](#)
- [recvfrom](#)
- [RIOCloseCompletionQueue](#)
- [RIOCreateCompletionQueue](#)
- [RIOCreateRequestQueue](#)
- [RIODequeueCompletion](#)
- [RIODeregisterBuffer](#)
- [RIONotify](#)
- [RIOReceive](#)
- [RIOReceiveEx](#)
- [RIORegisterBuffer](#)
- [RIOResizeCompletionQueue](#)
- [RIOResizeRequestQueue](#)
- [RIOSend](#)
- [RIOSendEx](#)
- [select](#)
- [send](#)
- [sendto](#)
- [setipv4sourcefilter](#)
- [setsockopt](#)

- [setsourcefilter](#)
- [shutdown](#)
- [socket](#)
- [TransmitPackets](#)
- [WSAAccept](#)
- [WSAAddressToStringW](#)
- [WSACleanup](#)
- [WSACloseEvent](#)
- [WSAConnect](#)
- [WSAConnectByList](#)
- [WSAConnectByNameW](#)
- [WSACreateEvent](#)
- [WSAEnumNameSpaceProvidersW](#)
- [WSAEnumNetworkEvents](#)
- [WSAEnumProtocolsW](#)
- [WSAEventSelect](#)
- [WSAGetLastError](#)
- [WSAGetOverlappedResult](#)
- [WSAhtonl](#)
- [WSAhtons](#)
- [WSAioctl](#)
- [WSAJoinLeaf](#)
- [WSALookupServiceBeginW](#)
- [WSALookupServiceEnd](#)
- [WSALookupServiceNextW](#)
- [WSANSPIoclt](#)
- [WSANtohl](#)
- [WSANtohs](#)
- [WSARecv](#)
- [WSARecvFrom](#)
- [LPFN\\_WSARECVMSG \(WSARecvMsg\)](#)
- [WSAResetEvent](#)
- [WSASend](#)
- [WSASendTo](#)
- [WSASetEvent](#)
- [WSASetLastError](#)
- [WSASetServiceW](#)
- [WSASocketW](#)
- [WSAStartup](#)
- [WSAStringToAddressW](#)
- [WSAWaitForMultipleEvents](#)

## Updated for Windows 8 and Windows Server 2012

Microsoft Windows 8 and Windows Server 2012 introduce new Windows Sockets programming elements.

A set of high-speed networking extensions are available for increased networking performance with lower latency and jitter. These extensions targeted primarily for server applications use pre-registered data buffers and

completion queues to increase performance.

The following are new Windows Sockets functions added to support Winsock high-speed networking Registered I/O extensions:

- [RIOCloseCompletionQueue](#)
- [RIOCreatCompletionQueue](#)
- [RIOCreatRequestQueue](#)
- [RIODequeueCompletion](#)
- [RIODeregisterBuffer](#)
- [RIONotify](#)
- [RIOReceive](#)
- [RIOReceiveEx](#)
- [RIORegisterBuffer](#)
- [RIOResizeCompletionQueue](#)
- [RIOResizeRequestQueue](#)
- [RIOSend](#)
- [RIOSendEx](#)

The following are new Windows Sockets enumerations, structures, and typedefs added to support Winsock high-speed networking Registered I/O extensions:

- [RIO\\_CQ](#)
- [RIO\\_RQ](#)
- [RIO\\_BUFFERID](#)
- [RIO\\_BUF](#)
- [RIO\\_NOTIFICATION\\_COMPLETION](#)
- [RIO\\_NOTIFICATION\\_COMPLETION\\_TYPE](#)
- [RIORESULT](#)

A set of enhancements for asynchronous naming support are available.

The following are new Windows Sockets functions added to support asynchronous naming requests:

- [GetAddrInfoExCancel](#)
- [GetAddrInfoExOverlappedResult](#)

The following existing Winsock functions have been revised to support asynchronous naming requests:

- [GetAddrInfoEx](#)
- [NSPv2LookupServiceBegin](#)

A set of enhancements to add support for Internationalized Domain Name (IDN) parsing are available.

The following existing Winsock functions have been revised to support IDN parsing:

- [getaddrinfo](#)
- [GetAddrInfoEx](#)
- [GetAddrInfoW](#)

An enhancement for naming support in Winsock to support requesting both a canonical name and a fully qualified domain name are available.

The following existing Winsock function has been revised to support requesting both a canonical name and a fully qualified domain name:

- [GetAddrInfoEx](#)

The following new Winsock structure has been added to support requesting both a canonical name and a fully qualified domain name:

- [addrinfoex2](#)

The following new Windows Sockets function has been added to retrieve the local host name in Unicode:

- [GetHostNameW](#)

The following are new Windows Sockets IOCTLs added to support the Windows Filtering Platform (WFP) redirect service:

- [SIO\\_QUERY\\_WFP\\_CONNECTION\\_REDIRECT\\_CONTEXT](#)
- [SIO\\_QUERY\\_WFP\\_CONNECTION\\_REDIRECT\\_RECORDS](#)
- [SIO\\_SET\\_WFP\\_CONNECTION\\_REDIRECT\\_RECORDS](#)

The following are new Windows IP socket options added to support the Windows Filtering Platform (WFP) redirect service:

- [IP\\_WFP\\_REDIRECT\\_CONTEXT](#)
- [IP\\_WFP\\_REDIRECT\\_RECORDS](#)

The following are a new Windows Sockets IOCTL and an associated structure added to control the initial (SYN / SYN+ACK) retransmission characteristics of a TCP socket:

- [SIO\\_TCP\\_INITIAL\\_RTO](#)
- [TCP\\_INITIAL\\_RTO\\_PARAMETERS](#)

The following are a new Windows Sockets IOCTL and an associated structure added to retrieve the association between a socket and an RSS processor core and NUMA node:

- [SIO\\_QUERY\\_RSS\\_PROCESSOR\\_INFO](#)
- [SOCKET\\_PROCESSOR\\_AFFINITY](#)

The following new Windows Sockets IOCTLs are added to apply and query transport settings on a socket:

- [SIO\\_APPLY\\_TRANSPORT\\_SETTING](#)
- [SIO\\_QUERY\\_TRANSPORT\\_SETTING](#)

The only transport setting currently defines is for the **REAL\_TIME\_NOTIFICATION\_CAPABILITY** capability on a TCP socket. The following new structures and enumerations are added to support the **REAL\_TIME\_NOTIFICATION\_CAPABILITY**:

- [CONTROL\\_CHANNEL\\_TRIGGER\\_STATUS](#)
- [REAL\\_TIME\\_NOTIFICATION\\_SETTING\\_INPUT](#)
- [REAL\\_TIME\\_NOTIFICATION\\_SETTING\\_OUTPUT](#)
- [TRANSPORT\\_SETTING\\_ID](#)

The following new Windows Sockets IOCTL is added to enable a fast path for loopback on a TCP socket. This feature can lower latency and improve performance for applications that use TCP loopback (applications used by the financial service industry, for example).:

- [SIO\\_LOOPBACK\\_FAST\\_PATH](#)

An enhancement to support transferring streaming media that require quality of service (Voice over IP, for

example).

The following new Windows Sockets function supports transferring streaming media that require quality of service:

- [SetSocketMediaStreamingMode](#)

The [SetSocketMediaStreamingMode](#) function is also supported for Windows Store apps on Windows 8, Windows Server 2012, and later.

A set of inline functions defined in the *Winsock2.h* header file for converting a `float` or an `unsigned __int64` between host byte order and network byte order.

- [hton](#)
- [htonf](#)
- [htonll](#)
- [ntohd](#)
- [ntohf](#)
- [ntohll](#)

## Updated for Windows 7 and Windows Server 2008 R2

Microsoft Windows 7 and Windows Server 2008 R2 introduce new Windows Sockets programming elements.

The following existing Windows Sockets functions were enhanced to support additional options:

- [getaddrinfo](#)
- [GetAddrInfoEx](#)
- [GetAddrInfoW](#)

The following are new Windows Sockets options:

- [IP\\_ORIGINAL\\_ARRIVAL\\_IF](#)
- [IP\\_ORIGINAL\\_ARRIVAL\\_IF for IPv6](#)

Dynamic send buffering for TCP was added on Windows 7 and Windows Server 2008 R2. As a result, the use of the `SIO_IDEAL_SEND_BACKLOG_CHANGE` and `SIO_IDEAL_SEND_BACKLOG_QUERY` IOCTLs are needed only in special circumstances. For more information, see [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#).

## Updated for Windows Server 2008 and Windows Vista with Service Pack 1 (SP1)

Microsoft Windows Server 2008 and Windows Vista with Service Pack 1 (SP1) introduce new Windows Sockets programming elements.

The following are new Windows Sockets IOCTLs:

- [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#)
- [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#)

These new IOCTLs can be used by an application using TCP to determine the ideal value for the amount of data outstanding to send in order to achieve the best throughput for a connection. This is called the ideal send backlog (ISB) size.

## Updated for Windows Server 2008

Microsoft Windows Server 2008 introduce new Windows Sockets programming elements.

The following are new Windows Sockets options:

- [SO\\_PORT\\_SCALABILITY](#)

## Updated for Windows Vista

Microsoft Windows Vista introduces new Windows Sockets programming elements. These elements extend the capability of Winsock to simplify programming and provide IPv6 compatibility. For more information about porting Winsock applications to IPv6, see [IPv6 Guide for Windows Sockets Applications](#).

Windows sockets tracing is a new feature supported on Windows Vista and Windows Server 2008. This feature can be enabled in retail binaries to trace certain socket events with minimal overhead. For more information, see [Winsock Tracing](#).

The following are new Windows Sockets functions:

- [FreeAddrInfoEx](#)
- [GetAddrInfoEx](#)
- [InetNtop](#)
- [InetPton](#)
- [SetAddrInfoEx](#)
- [WSAConnectByList](#)
- [WSAConnectByName](#)
- [WSADeleteSocketPeerTargetName](#)
- [WSAEnumNameSpaceProvidersEx](#)
- [WSAImpersonateSocketPeer](#)
- [WSAPoll](#)
- [WSAQuerySocketSecurity](#)
- [WSARevertImpersonation](#)
- [WSASendMsg](#)
- [WSASetSocketPeerTargetName](#)
- [WSASetSocketSecurity](#)

The following are new Windows Sockets structures and enumerations:

- [addrinfoex](#)
- [BLOB](#)
- [GROUP\\_FILTER](#)
- [GROUP\\_REQ](#)
- [GROUP\\_SOURCE\\_REQ](#)
- [MULTICAST\\_MODE\\_TYPE](#)
- [NAPI\\_DOMAIN\\_DESCRIPTION\\_BLOB](#)
- [NAPI\\_PROVIDER\\_INSTALLATION\\_BLOB](#)
- [NAPI\\_PROVIDER\\_LEVEL](#)
- [NAPI\\_PROVIDER\\_TYPE](#)
- [SOCKET\\_PEER\\_TARGET\\_NAME](#)
- [SOCKET\\_SECURITY\\_PROTOCOL](#)
- [SOCKET\\_SECURITY\\_QUERY\\_INFO](#)
- [SOCKET\\_SECURITY\\_QUERY\\_TEMPLATE](#)
- [SOCKET\\_SECURITY\\_SETTINGS](#)
- [SOCKET\\_SECURITY\\_SETTINGS\\_IPSEC](#)

- [SOCKET\\_USAGE\\_TYPE](#)
- [WSAQUERYSET2](#)

Microsoft Windows Vista introduces new Windows Sockets SPI functions to provide the ability to categorize applications for a layered service providers. New functions are also added for namespace providers.

The following are new Windows Sockets SPI functions:

- [NSPv2Cleanup](#)
- [NSPv2ClientSessionRundown](#)
- [NSPv2LookupServiceBegin](#)
- [NSPv2LookupServiceEnd](#)
- [NSPv2LookupServiceNextEx](#)
- [NSPv2SetServiceEx](#)
- [NSPv2Startup](#)
- [WSAAdvertiseProvider](#)
- [WSAProviderCompleteAsyncCall](#)
- [WSAUnadvertiseProvider](#)
- [WSCEnumNameSpaceProvidersEx32](#)
- [WSCGetApplicationCategory](#)
- [WSCGetProviderInfo](#)
- [WSCInstallNameSpaceEx](#)
- [WSCInstallNameSpaceEx32](#)
- [WSCSetApplicationCategory](#)
- [WSCSetProviderInfo](#)
- [WSCSetProviderInfo32](#)

The following are new Windows Sockets SPI structures:

- [NSPV2\\_ROUTINE](#)

Microsoft Windows Vista introduces new Windows Sockets programming elements.

The following are new Windows Sockets ioctlS:

- [SIO\\_ACQUIRE\\_PORT\\_RESERVATION](#)
- [SIO\\_ASSOCIATE\\_PORT\\_RESERVATION](#)
- [SIO\\_SET\\_COMPATIBILITY\\_MODE](#)
- [SIO\\_RELEASE\\_PORT\\_RESERVATION](#)

The following are new Windows Sockets options:

- [SO\\_BSP\\_STATE](#)
- [IP\\_UNICAST\\_IF](#)
- [IPV6\\_UNICAST\\_IF](#)

## Updated for April 2005

The following functions have been added to the Windows Sockets SPI (ws2spi.h) to support 32-bit processes and catalogs on 64-bit platforms:

- [WSCDeinstallProvider32](#)
- [WSCEnableNSProvider32](#)
- [WSCEnumNameSpaceProviders32](#)

- [WSCEnumProtocols32](#)
- [WSCGetProviderPath32](#)
- [WSCInstallNameSpace32](#)
- [WSCInstallProvider64\\_32](#)
- [WSCInstallProviderAndChains64\\_32](#)
- [WSCUnInstallNameSpace32](#)
- [WSCUpdateProvider32](#)

## Updated for Windows Server 2003

Microsoft Windows Server 2003 introduces new Windows Sockets programming elements. These elements extend the capability of Winsock to simplify programming and provide IPv6 compatibility. For more information about porting Winsock applications to IPv6, see [IPv6 Guide for Windows Sockets Applications](#).

The following are new Windows Sockets functions:

- [ConnectEx](#)
- [DisconnectEx](#)
- [freeaddrinfo](#)
- [gai\\_strerror](#)
- [getaddrinfo](#)
- [getnameinfo](#)
- [TransmitPackets](#)
- [WSANSPIoclt](#)
- [LPFN\\_WSARECVMSG \(WSARecvMsg\)](#)

The following are new Windows Sockets structure definitions:

- [addrinfo](#)
- [in\\_pktinfo](#)
- [SOCKADDR\\_STORAGE](#)
- [TRANSMIT\\_PACKETS\\_ELEMENT](#)
- [WSAMSG](#)

# Winsock Network Protocol Support in Windows

3/5/2021 • 4 minutes to read • [Edit Online](#)

The Internet Protocol Suite is the dominant network protocol used in enterprise networks and across the Internet. The Internet Protocol Suite represents a large collection of layered network protocols. The Internet Protocol Suite is often referred to as TCP/IP based on two of the most important protocols included in the suite: the Internet Protocol (IP) and the Transmission Control Protocol (TCP).

IPv6 and IPv4 represent the two available versions of the Internet Protocol. TCP is one of several important network services often referred to as IP protocols that operate over IPv6 and IPv4 networks. The User Datagram Protocol (UDP) and Internet Control Message Protocol (ICMP) are other important IP protocols used over IPv6 and IPv4 networks. There are a number of other IP protocols that can be used over IPv6 and IPv4 networks.

Windows Sockets considers each network protocol suite as a unique address family. So the IPv6 protocol is considered the **AF\_INET6** address family and the IPv4 protocol is considered the **AF\_INET** address family. The IPv6 and IPv4 protocols support the use of various layered IP protocols such as TCP, UDP, and ICMP.

Windows Sockets were initially designed to add support for IPv4 to Windows. However, the Windows Sockets programming interface was designed from the onset with the ability to support other network protocol suites. Over time, versions of Windows and the associated Windows Sockets have included native support for other network protocol suites (IPX/SPX and AppleTalk, for example). Support for other network protocols was also available for versions of Windows as third-party software from vendors.

Before the growth and popularity of the Internet, various other network protocol suites were used in networked environments, particularly for local Intranets. The choice of a network protocol suite was often based on the size of the network or the expertise of the IT networking staff. With today's global Internet connectivity linking even the smallest networks to the rest of the world, networking expertise in IPv6 and IPv4 is essential for networking professionals. As a result, other previously important network protocol suites are now in very limited use and have become obviated. Native support for these obviated network protocol suites, often referred to as legacy network protocols, has been dropped from recent versions of Microsoft Windows. Support for some of these legacy protocols may be available as third-party software from vendors (ATM with ATM network hardware, for example).

The following table identifies native Windows support for common network protocol suites.

NETWORK PROTOCOL	WINDOWS 7	WINDOWS SERVER 2008	WINDOWS VISTA	WINDOWS SERVER 2003	WINDOWS XP	WINDOWS 2000
IPX/SPX	Not supported	Not supported	Not supported	Supported	Supported	Supported
AppleTalk	Not supported	Not supported	Not supported	Supported	Supported	Supported
DLC	Not supported	Not supported	Not supported	Not supported (see Notes)	Not supported (see Notes)	Supported
ATM	Not supported	Not supported	Not supported	Supported (see Notes)	Supported (see Notes)	Supported (see Notes)
NetBEUI	Not supported	Not supported	Not supported	Not supported	Not supported	Supported (see Notes)

**IPv6 on Windows 2000:** The IPv6 protocol is supported on Windows 2000 with Service Pack 1 (SP1) and later with the Microsoft IPv6 Technology Preview for Windows 2000.

**NetBIOS:** The NetBIOS protocol is commonly used by naming services on Windows. NetBIOS can use multiple network protocol suites including IP (NetBIOS over TCP/IP), IPX/SPX, and NetBEUI. Winsock supports NetBIOS over TCP/IP (commonly call NetBT) only on the 32-bit versions of Windows 7, Windows Server 2008, and Windows Vista. Winsock supports NetBIOS over TCP/IP and NetBIOS using IPX on Windows Server 2003 and Windows XP. Winsock supports NetBIOS over TCP/IP, NetBIOS using IPX, and NetBIOS using NetBEUI on Windows 2000.

**IrDA:** The Infrared Data Association (IrDA) protocol is supported if the computer has an infrared port and driver installed.

**Bluetooth:** Winsock support for Bluetooth as a network protocol suite includes the Bluetooth Personal Area Network (PAN) and Dial up Networking (DUN) profiles. Bluetooth support in Windows also includes using the Bluetooth Human Interface Device (HID) and other profiles for connecting to keyboards, pointing devices, and other input devices which are unrelated to network protocols.

**DLC on Windows 2003 and Windows XP:** The Data Link Control (DLC) protocol is supported on Windows Server 2003 and Windows XP when the DLC driver included with Microsoft Host Integration Server 2006, Host Integration Server 2004, or Host Integration Server 2000 is installed.

**ATM on Windows 2003, Windows XP, and Windows 2000:** The Asynchronous Transfer Mode (ATM) protocol is supported on Windows Server 2003, Windows XP, and Windows 2000 when an ATM network adapter is installed. The protocol for classical IP over ATM (sometimes abbreviated as CLIP/ATM) is defined in [RFC 2225](#) and related documents published by the IETF. Windows Server 2003, Windows XP, and Windows 2000 provide a full implementation of this standard.

**NetBEUI on Windows 2000:** The NetBEUI protocol is not directly supported by Windows sockets. But the NetBIOS protocol which may use multiple network protocols supports using the NetBEUI protocol on Windows 2000.

## Related topics

[ATM Technical Reference](#)

Bluetooth

IPv6 Technology Preview for Windows 2000

IrDA

NDIS 5.0 and ATM Support in Windows

# About Winsock

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section describes Winsock programming considerations, architecture, and capabilities available to Winsock developers. The following list describes the topics in this section:

- [Winsock Programming Considerations](#)
- [Simultaneous Access to Multiple Transport Protocols](#)
- [Overlapped I/O and Event Objects](#)
- [Registration and Name Resolution](#)
- [Multipoint and Multicast Semantics](#)
- [Internet Protocol Version 6 \(IPv6\)](#)
- [Network Location Awareness Service Provider \(NLA\)](#)
- [About the Winsock SPI](#)

## Related topics

[Using Winsock](#)

[What's New for Windows Sockets](#)

[Winsock Network Protocol Support in Windows](#)

[Winsock Reference](#)

# Winsock Programming Considerations

3/5/2021 • 2 minutes to read • [Edit Online](#)

Windows Sockets 2 extends the functionality of Windows Sockets 1.1 in a number of areas. The following table summarizes some of the major feature changes.

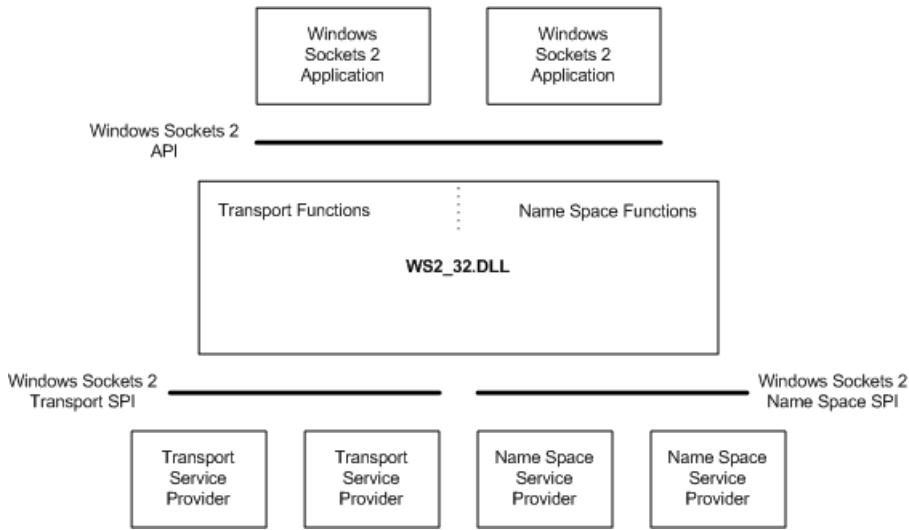
FEATURES	DESCRIPTION
<a href="#">Windows Sockets 2 Architecture</a>	A description of the Windows Sockets 2 architecture.
<a href="#">Socket handles</a>	A socket handle can optionally be a file handle in Windows Sockets 2. It is possible to use socket handles with standard Windows file I/O functions.
<a href="#">Simultaneous access to multiple transport protocols</a>	Allows an application to use the familiar socket interface to achieve simultaneous access to a number of installed transport protocols.
<a href="#">Protocol-independent name resolution</a>	Includes a standardized set of functions for querying and working with the myriad name resolution domains that exist today (for example DNS, SAP, and X.500).
<a href="#">Protocol-independent multicast and multipoint</a>	Applications discover what type of multipoint or multicast capabilities a transport provides and use these facilities in a generic manner.
<a href="#">Overlapped I/O</a>	Incorporates the overlapped paradigm for socket I/O following the model established in Windows environments.
<a href="#">Scatter/gather I/O</a>	Incorporates scatter/gather capabilities with the overlapped paradigm for socket I/O, following the model established in Windows environments.
<a href="#">Quality of Service (QoS)</a>	Establishes conventions that applications use to negotiate required service levels for parameters such as bandwidth and latency. Other QoS-related enhancements include mechanisms for network-specific Quality of Service extensions.
<a href="#">Provider-Specific Extension Mechanism</a>	The <a href="#">WSAIoctl</a> function enables service providers to offer provider-specific feature extensions.
<a href="#">Shared Sockets</a>	The <a href="#">WSADuplicateSocket</a> function is introduced to enable socket sharing across processes.
<a href="#">Connection Setup and Teardown</a>	An application can obtain caller information such as caller identifier and Quality of Service before deciding whether to accept an incoming connection request. It is also possible (for protocols that support this) to exchange user data between the endpoints at connection teardown time.
<a href="#">Graceful Shutdown, Linger Options, and Socket Closure</a>	An application has several options for shutting down a socket connection (shutdown sequence).

FEATURES	DESCRIPTION
<a href="#">Protocol-Independent Out-of-Band Data</a>	The stream socket abstraction includes the notion of out of band (OOB) data.
<a href="#">Debug and Trace Facilities</a>	Windows Sockets 2 supports a specially devised version of the Ws2_32.dll and a separate debug/trace DLL.
<a href="#">Windows Sockets Compatibility Issues</a>	Windows Sockets 2 continues to support all of the Windows Sockets 1.1 semantics and function calls except for those dealing with pseudo-blocking.
<a href="#">Handling Winsock Errors</a>	How Winsock errors can be retrieved and handled by an application.

# Windows Sockets 2 Architecture

3/5/2021 • 2 minutes to read • [Edit Online](#)

The Windows Sockets 2 architecture is compliant with the Windows Open System Architecture (WOSA), as illustrated below:



Winsock 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, Winsock support is not limited to TCP/IP protocol stacks as is the case for Windows Sockets 1.1.

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.

# Socket Handles

3/5/2021 • 2 minutes to read • [Edit Online](#)

A socket handle can optionally be a file handle in Windows Sockets 2. A socket handle from a Winsock provider can be used with other non-Winsock functions such as [ReadFile](#), [WriteFile](#), [ReadFileEx](#), and [WriteFileEx](#).

The `XP1_IFS_HANDLES` member in the protocol information structure for a provider determines whether a socket handle from a provider is an Installable File System (IFS) handle. Socket handles that are IFS handles can be used without a performance penalty with other non-Winsock functions ([ReadFile](#) and [WriteFile](#), for example). Any non-IFS socket handles when used with non-Winsock functions ([ReadFile](#) and [WriteFile](#), for example) result in interactions between the provider and the file system where extra processing overhead is involved that can result in a significant performance penalty. When using socket handles with non-Winsock functions, the error codes propagated from the base file system are not always mapped to Winsock error codes. Consequently, it is recommended that socket handles be used only with Winsock functions.

A socket handle should not be used with the [DuplicateHandle](#) function. The presence of layered service providers (LSPs) can cause this to fail and there is no way for the destination process to import the socket handle.

## NOTE

Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use [Windows Filtering Platform](#).

Windows Sockets 2 has expanded certain functions that transfer data between sockets using handles. The functions offer advantages specific to sockets for transferring data and include [WSARecv](#), [WSASend](#), and [WSADuplicateSocket](#).

# Scatter/Gather I/O

3/5/2021 • 2 minutes to read • [Edit Online](#)

The [WSARecv](#), [WSARecvFrom](#), [LPFN\\_WSARECVMSG \(WSARecvMsg\)](#), [WSASend](#), [WSASendMsg](#), and [WSASendTo](#) 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 the 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 by itself.

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

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 set up, utilize, and tear down multipoint sessions.

The following summarizes Winsock features 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 [WSA\\_IOCTL](#) 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.

# Flow Specification Quality of Service

3/5/2021 • 2 minutes to read • [Edit Online](#)

Quality of Service is implemented in Windows through various system QoS components. For complete details and implementation guidelines, see [Quality of Service](#).

For details about QoS templates, see [QoS Templates](#).

For details and implementation guidelines about Quality of Service, see the [FLOWSPEC](#) structure.

# Provider-Specific Extension Mechanism

3/5/2021 • 2 minutes to read • [Edit Online](#)

The [WSAIoctl](#) function enables service providers to offer provider-specific feature 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.

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 [WSAIoctl](#) 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.

On Windows Vista and later, new Winsock system extensions are exported directly from the Winsock DLL, so the [WSAIoctl](#) function is not needed to load these extensions. The new extension functions available on Windows Vista and later include the [WSAPoll](#) and [WSASendMsg](#) functions that are exported from *Ws2\_32.dll*.

# Shared Sockets

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 identifier. 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 Winsock 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, remains open until [closesocket](#) is called with the last remaining descriptor.

Notification on shared sockets is subject to the usual constraints of the [WSAAAsyncSelect](#) and [WSAEEventSelect](#) 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.

# Connection Setup and Teardown

3/5/2021 • 2 minutes to read • [Edit Online](#)

The [WSAAccept](#) function lets an application obtain caller information such as caller identifier and Quality of Service 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 teardown is the delivery of disconnect data from the teardown initiator. After receiving notice that the remote end has initiated teardown (typically by the FD\_CLOSE indication), the [WSARecvDisconnect](#) function can be called to receive the disconnect data, if any.

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

CLIENT SIDE	SERVER SIDE
(1) Invoke <a href="#">WSASendDisconnect</a> to conclude session and supply transaction total.	
	(2) Get FD_CLOSE, <a href="#">recv</a> with a return value of zero, or <a href="#">WSAEDISCON</a> error return from <a href="#">WSARecv</a> indicating graceful shutdown in progress.
	(3) Invoke <a href="#">WSARecvDisconnect</a> to get client's transaction total.
	(4) Compute cumulative grand total of all transactions.
	(5) Invoke <a href="#">WSASendDisconnect</a> to transmit grand total.
(6) Receive FD_CLOSE indication.	(5a) Invoke <a href="#">closesocket</a> .
(7) Invoke <a href="#">WSARecvDisconnect</a> to receive and store cumulative grand total of transactions.	
(8) Invoke <a href="#">closesocket</a>	

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



# Graceful Shutdown, Linger Options, and Socket Closure

3/5/2021 • 3 minutes to read • [Edit Online](#)

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 implicitly causes 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 to 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 are still not 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.

The use of the SO\_LINGER and SO\_DONTLINGER socket options and the associated [linger](#) structure is discussed in more detail in the reference sections on [SOL\\_SOCKET Socket Options](#) and the [linger](#) structure.

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 causes an FD\_CLOSE indication to be received by the peer application indicating that all pending data has been received. To illustrate this, the following table shows the functions that would be invoked by the client and server components of an application, where the client is responsible for initiating a graceful shutdown.

CLIENT SIDE	SERVER SIDE
(1) Invokes <code>shutdown(s, SD_SEND)</code> to signal end of session and that client has no more data to send.	
	(2) Receives FD_CLOSE, indicating graceful shutdown in progress and that all data has been received.
	(3) Sends any remaining response data.
(local timing significance only) Gets FD_READ and calls <code>recv</code> to get any response data sent by server .	(4) Invokes <code>shutdown(s, SD_SEND)</code> to indicate server has no more data to send.
(5) Receives FD_CLOSE indication.	(local timing significance only) Invokes <code>closesocket</code> .
(6) Invokes <code>closesocket</code> .	

# Protocol-Independent Out-of-Band Data

3/5/2021 • 6 minutes to read • [Edit Online](#)

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 following section describes OOB data handling in a protocol-independent manner. A discussion of OOB data implemented using TCP urgent data follows the protocol-independent explanation. In each discussion, the use of `recv` also implies `recvfrom`, `WSARecv`, and `WSARecvFrom`, and references to `WSAAAsyncSelect` also apply to `WSAEEventSelect`.

## 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 (OOB) data.

A user can determine if any OOB data is waiting to be read using the `ioctlsocket` or `WSAIoctl` function with the `SIOCATMARK` IOCTL. 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 maintains 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` or `WSAIoctl` functions that support `SIOCATMARK`. The presence or absence of OOB data is all 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 inline, as part of the normal data stream. This is achieved by setting the socket option `SO_OOBINLINE` with the `setsockopt` function. For other protocols where the OOB data blocks are truly independent of the normal data stream, attempting to set `SO_OOBINLINE` results in an error. An application can use the `ioctlsocket` or `WSAIoctl` function with the `SIOCATMARK` IOCTL to determine whether there is any unread OOB data preceding the mark. For example, it can use this information 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 `WSAAAsyncSelect`, 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 does 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 does not span that position. One `recv` returns 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 `WSAAAsyncSelect` functions, and indicated by setting the socket in `readfds` or by sending an `FD_READ` message respectively.
- The application cannot call `recv` with the `MSG_OOB` flag set to read the OOB data block. The error code `WSAEINVAL` is returned.
- The application can call `recv` without the `MSG_OOB` flag set. Any OOB data is delivered in its correct order within the normal data stream. OOB data is never 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 `WSAAAsyncSelect` 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 data (OOB), implemented using TCP urgent data, follows the model used in the Berkeley software distribution. Users and implementers should be aware that:

- There are, at present, two conflicting interpretations of [RFC 793](#) (where the concept is introduced).
- The implementation of OOB data in the Berkeley Software Distribution (BSD) does not conform to the Host Requirements specified 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 implementation compatible with RFC 1122, the receiver reads the wrong urgent data byte (it reads 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 OOB data unless this is required to interoperate with an existing service. Windows Sockets suppliers are urged to document the OOB semantics (BSD or RFC 1122) that their product implements.

The 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 in the preceding). 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.

**NOTE**

When the SO\_OOBINLINE socket option is set, the SIOCATMARK IOCTL always returns **TRUE**, and OOB data is returned to the user as normal data.

# Debug and Trace Facilities

3/5/2021 • 3 minutes to read • [Edit Online](#)

Windows Sockets 2 application developers need to isolate bugs in:

- The application.
- The *Ws2\_32.dll* or one of the compatibility shim DLLs.
- The service provider.

Windows Sockets 2 addresses this need through several components and features:

- Integrated support for Winsock tracing on Windows Vista and later.
- A specially devised debug version of the *Ws2\_32.dll* on Windows Vista.
- A separate primitive debug and trace facility for use on Windows Server 2003 and Windows XP.

## Winsock Tracing using Event Tracing for Windows

Integrated support for Winsock tracing using Event Tracing for Windows (ETW) is included on Windows Vista and later. This is the preferred method for tracing Winsock calls on Windows Vista and later. Winsock tracing using ETW is lightweight and works on retail versions of Windows. No additional software or components are required. This feature just needs to be enabled on Windows Vista and later. For more detailed information, see the [Winsock Tracing](#) topics.

## Using a Debug Version of *Ws2\_32.dll*

The combination of a debug version of the *Ws2\_32.dll* on Windows Vista and Winsock tracing allows all procedure calls across the Windows Sockets 2 API or SPI to be monitored and, to some extent, controlled.

If a version of the Microsoft Windows Software Development Kit (SDK) for Windows Vista is installed to the default location, debug versions of the *Ws2\_32.dll* for various architectures are located under the following folder:

**C:\Program Files\Microsoft SDKs\Windows\v6.0\NoRedist**

A checked version of the *Ws2\_32.dll* that matches the version of Windows and the Service Pack you are testing on should be used. Be aware that security patches may have been applied that updated the *Ws2\_32.dll* on your test system. The Windows SDK for Windows Vista and the earlier Platform Software Development Kit (SDK) DVD/CD subscriptions include checked builds for the various versions of Windows. You should use the same checked version of the *Ws2\_32.dll* as the retail version that was used on the system being tested. Note also that behavior running under a checked build will not be the same as running with a retail build.

**Note** The Windows SDK for Windows Server 2008 and later no longer includes special debug versions of the *Ws2\_32.dll*. Developers should use Winsock tracing using ETW instead, since this feature does not require debug builds.

## Winsock Debug and Trace Facility on Windows Server 2003 and Windows XP

Older versions of Windows prior to Windows 8 and Windows Server 2012 support a separate primitive debug and trace facility that is included as a sample with the Windows SDK and the older Platform SDK. The debug/trace facility should only be used on Windows Server 2003 and Windows XP where Winsock tracing is

not supported.

If the Windows SDK for Windows 7 is installed to the default location, this primitive Winsock tracing feature is installed in the following folder:

C:\Program Files\Microsoft SDKs\Windows\v7.0\Samples\NetDs\winsock\dt\_dll

The *DbgSpec.doc* file in this folder provides documentation on this primitive trace facility. The sample code in the dt\_dll folder needs to be compiled to use this facility. Developers are free to use the source code to develop versions of the debug/trace DLL that meet their specific needs.

Note that this primitive Winsock trace feature will only work with the debug version of *Ws2\_32.dll* installed. So you will need to get a checked version of the *Ws2\_32.dll* that matches the version of Windows and the Service Pack you are testing on.

A limitation of this primitive dt\_dll trace facility is that the sample code uses a global lock (critical section) for each Winsock function call. So this facility is not useful in dealing with race conditions. The sample code would need to be substantially rewritten to make this trace facility useful for dealing with most real Winsock problems (replacing the global locks). This sample code does allow developers to trace the procedure calls, procedure returns, parameter values, and return values.

Developers can use this primitive 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 is better able to isolate a problem in the application, *Ws2\_32.dll*, or service provider.

## Related topics

[Winsock Tracing](#)

# Windows Sockets Compatibility Issues

3/5/2021 • 2 minutes to read • [Edit Online](#)

Windows Sockets 2 continues to support all of the Windows Sockets 1.1 semantics and function calls except for those dealing with pseudo-blocking. Since Windows Sockets 2 runs only in 32-bit, preemptively scheduled environments, there is no need to implement the pseudo-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 pseudo-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 code modification must occur. In most cases, the judicious use of threads can be substituted to accommodate processing that was being accomplished with a blocking hook function.

# Default State for a Socket's Overlapped Attribute

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 ensure backward compatibility with currently deployed Wsock32.dll implementations, this will continue to be the case for Windows Sockets 2 as well. That is, in Windows Sockets 2 sockets created with the `socket` function will have the overlapped attribute. However, in order to be more compatible with the rest of the Windows API, sockets created with `WSASocket` will not, default, have the overlapped attribute. This attribute will only be applied if the `WSA_FLAG_OVERLAPPED` bit is set.

# Windows Sockets 1.1 Blocking Routines and EINPROGRESS

3/5/2021 • 4 minutes to read • [Edit Online](#)

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](#) function, 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 blocking mode unless the programmer explicitly requests that operations be treated as nonblocking. Windows Sockets 1.1 environments could not assume preemptive 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 pseudo-blocking facilities described in the following were provided.

## NOTE

Windows Sockets 2 only runs on preemptive 32-bit operating systems where deadlocks are not a problem. Programming practices 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 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. The following functions can block:

- [recv](#)
- [recvfrom](#)
- [send](#)
- [sendto](#)

With 16-bit Windows Sockets 1.1, a blocking operation that cannot complete immediately is handled by pseudo-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 call. Only one outstanding function call is allowed for a particular task. The only exceptions are two functions that are provided to assist the programmer in this situation: [WSAIIsBlocking](#) and [WSACancelBlockingCall](#).

The [WSAIIsBlocking](#) 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 fails with the error WSAEINPROGRESS.

It should be emphasized that this restriction applies to both blocking and nonblocking 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 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 can be called instead of the default message dispatch routine described in the preceding discussion.

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.
- The request cannot be completed immediately.

A socket is set to blocking by default, but the [ioctlsocket](#) function with the FIONBIO IOCTL or the [WSAAAsyncSelect](#) function can set a socket to nonblocking mode.

The blocking hook is never 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 guidelines:

- It uses only nonblocking sockets.
- It uses the [WSAAAsyncSelect](#) and/or the [WSAAAsyncGetXByY](#) 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 address viability of the memory involved.

# Handling Winsock Errors

3/5/2021 • 2 minutes to read • [Edit Online](#)

Most Windows Sockets 2 functions do not return the specific cause of an error when the function returns. Some Winsock functions return a value of zero if successful. Otherwise, the value SOCKET\_ERROR (-1) is returned and a specific error number can be retrieved by calling the [WSAGetLastError](#) function. For Winsock functions that return a handle, a return value of INVALID\_SOCKET (0xffff) indicates an error and a specific error number can be retrieved by calling [WSAGetLastError](#). For Winsock functions that return a pointer, a return value of **NULL** indicates an error and a specific error number can be retrieved by calling the [WSAGetLastError](#) function.

A Winsock error code can be converted to an HRESULT for use in a remote procedure call (RPC) using `HRESULT_FROM_WIN32`. In earlier versions of the Platform Software Development Kit (SDK), `HRESULT_FROM_WIN32` was defined as a macro in the *Winerror.h* header file. In the Microsoft Windows Software Development Kit (SDK), `HRESULT_FROM_WIN32` is defined as an inline function in the *Winerror.h* header file.

## Related topics

[Windows Sockets Error Codes](#)

# Simultaneous Access to Multiple Transport Protocols

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 library 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. For more information, see [Windows Sockets 2 SPI](#). Any transport stack installed in this manner is referred to as a Windows Sockets service provider.

On Windows XP with Service Pack 2 (SP2), Windows Server 2003 with Service Pack 1 (SP1), and Windows Vista and later, the Winsock catalog that contains a list of installed transport and namespace providers can be displayed in a command prompt with the following command:

```
netsh winsock show catalog
```

The Microsoft Windows Software Development Kit (SDK) includes *Sporder.exe*, which enables the user to view and modify the order in which service providers are enumerated. Using *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* application uses exported functions from *Sporder.dll* to reorder the service providers. As a result, installation applications can use the interface provided by *Sporder.dll* to programmatically reorder service providers.

- [Layered Protocols and Protocol Chains](#)
- [Using Multiple Protocols](#)
- [Multiple Provider Restrictions on Select](#)

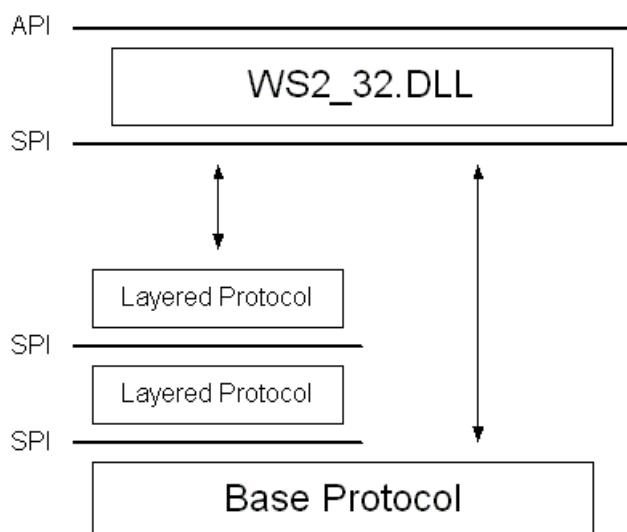
# Layered Protocols and Protocol Chains

3/5/2021 • 2 minutes to read • [Edit Online](#)

Windows Sockets 2 incorporates the concept of a layered protocol: 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 and 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.

You can create a protocol chain if you design the layered protocols to support the Windows Sockets 2 SPI at both their upper and lower edges. A special **WSAPROTOCOL\_INFO** structure refers to the protocol chain as a whole and describes the explicit order in which the layered protocols are joined. This is illustrated in the figure below. 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.



# Using Multiple Protocols

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 them 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) for IPv4 that consisted of a small number of well-known socket types and protocol identifiers. Windows Sockets 2 retains the existing address family, socket type, and protocol identifiers for compatibility reasons, but it also supports new address family values for new transport protocols with new media types.

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* parameters. 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

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 a set of sockets that spans more than one service provider:

- Using the **WSAWaitForMultipleEvents** or **WSAEventSelect** functions when blocking semantics are employed.
- Using the **WSAAAsyncSelect** function when nonblocking operations are employed and the application is already using a Windows message pump.

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 **WSAAAsyncSelect** function is not restricted to a single provider because it takes a single socket descriptor as an input parameter. Note however, that **WSAEventSelect** function offers better performance and scalability over **WSAAAsyncSelect** as the overhead of maintaining the message pump with Winsock event messages increases as the total number of sockets being used increases.

# Overlapped I/O and Event Objects

3/5/2021 • 3 minutes to read • [Edit Online](#)

Windows Sockets 2 supports overlapped I/O and all transport providers support this capability. Overlapped I/O follows the model established in Windows and can be performed on sockets created with the [socket](#) function or sockets created with the [WSASocket](#) function with the **WSA\_FLAG\_OVERLAPPED** flag set in the *dwFlags* parameter.

## NOTE

Creating a socket with the overlapped attribute has no impact on whether a socket is currently in 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 not to 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 can be filled in the order they are supplied, but the completion indications might occur in a different order.

In many cases, Winsock overlapped operations using [AcceptEx](#), [ConnectEx](#), [WSASend](#), [WSARecv](#), [TransmitFile](#), and similar functions are cancelable. However, behavior is undefined for the continued use of a socket that has canceled outstanding operations. The [closesocket](#) function should be called after canceling an overlapped operation. Therefore, for best results, instead of canceling the I/O directly, the [closesocket](#) function should be called to close the socket which will eventually discontinue all pending operations.

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

# Event Objects (Windows Sockets 2)

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 Windows 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 Windows functions, using the same function name, but without the WSA prefix.

# Receiving Completion Indications

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 Windows implementations, the process or thread truly blocks. Since Windows Sockets 2 event objects are implemented as Windows events, the native Windows 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 is called after a successfully initiated overlapped I/O operation completes (successfully or otherwise). The completion routine follows the same rules as stipulated for Windows file I/O completion routines. That is, the completion routine is not 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>	<b>HEVENT</b>	<i>LPCOMPLETIONROUTINE</i>	COMPLETION INDICATION
---------------------	---------------	----------------------------	-----------------------

<i>LPOVERLAPPED</i>	<i>HEVENT</i>	<i>LPCOMPLETIONROUTINE</i>	COMPLETION INDICATION
NULL	Not applicable	Ignored	Operation completes synchronously. It behaves as if it were a nonoverlapped socket.
!NULL	NULL	NULL	Operation completes overlapped, but there is no Windows Sockets 2-supported completion mechanism. The completion port mechanism (if supported) can be used in this case. Otherwise, there is 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

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 [WSAAyncSelect](#) 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.

# Registration and Name Resolution

3/5/2021 • 2 minutes to read • [Edit Online](#)

Windows Sockets 2 is a set of functions that standardizes the way applications access and use the various network naming services. When using these functions, applications need not distinguish 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, 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 [Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 SPI](#).

This section describes registration and name resolution capabilities available to Winsock developers. The following list describes the topics in this section:

- [Protocol-Independent Name Resolution](#)
- [Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 API](#)

## Related topics

[About Winsock](#)

[Using Winsock](#)

# Protocol-Independent Name Resolution

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 ( service) to register its existence within (or become accessible to) one or more namespaces.
- The ability of the client application to find the service within a namespace 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 in the topics in this section.

This section describes protocol-independent name resolution capabilities available to Winsock developers. The following list describes the topics in this section:

- [Name Resolution Model](#)
- [Summary of Name Resolution Functions](#)
- [Name Resolution Data Structures](#)

## Related topics

[Registration and Name Resolution](#)

# Name Resolution Model

3/5/2021 • 6 minutes to read • [Edit Online](#)

A *namespace* refers to some capability to associate (as a minimum) the protocol and addressing attributes of a network service with one or more friendly names. Many namespaces are currently in wide use, including the Internet's [Domain Name System](#) (DNS), [Active Directory Domain Services](#), the bindery, NetWare Directory Services (NDS) from Novell, and X.500. These namespaces vary widely in how they are organized and implemented. Some of their properties are particularly important to understand from the perspective of Winsock name resolution.

## Types of Namespaces

There are three different types of namespaces in which a service can be registered:

- Dynamic
- Static
- Persistent

Dynamic namespaces allow services to register with the namespace on the fly, and for clients to discover the available services at run-time. Dynamic namespaces frequently rely on broadcasts to indicate the continued availability of a network service. Older examples of dynamic namespaces include the Service Advertising Protocol (SAP) namespace used within a NetWare environment and the Name Binding Protocol (NBP) namespace used by AppleTalk. The Peer Name Resolution Protocol (PNRP) namespace used on Windows is a more recent example of a dynamic namespace.

Static namespaces require all of the services to be registered ahead of time, that is, when the namespace is created. An example of a static namespace are the *hosts*, *protocol*, and *services* files used by most TCP/IP implementations. On Windows, these files are typically located in the *C:\windows\system32\drivers\etc* folder.

Persistent namespaces allow services to register with the namespace on the fly. Unlike dynamic namespaces however, persistent namespaces retain the registration information in nonvolatile storage where it remains until such time as the service requests that it be removed. Persistent namespaces 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. DNS is another example of a persistent namespace. Although there is a programmatic way to resolve DNS names using Windows Sockets, the DNS namespace provider in Windows does not support registering new DNS names using Winsock. You must use the DNS functions directly to register DNS names. For more information, see the [DNS Reference](#).

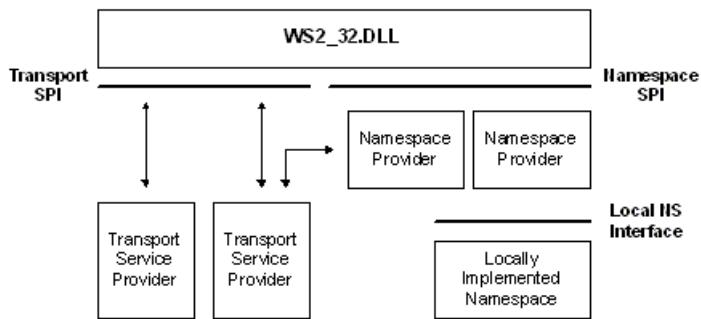
## Namespace Organization

Many namespaces 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 namespace hierarchy from which the search will begin.

## Namespace Provider Architecture

Naturally, the programmatic interfaces used to query the various types of namespaces and to register

information within a namespace (if supported) differ widely. A *namespace provider* is a locally-resident piece of software that knows how to map between the Winsock namespace SPI and some existing namespace (which could be implemented locally or be accessed through the network). Namespace provider architecture is illustrated as follows:



Note that it is possible for a given namespace, say DNS, to have more than one namespace provider installed on a given computer.

As mentioned above, the generic term *service* refers to the server-half of a client/server application. In Winsock, a service is associated with a *service class*, and each instance of a particular service has a *service name* which must be unique within the service class. Examples of service classes include FTP Server, SQL Server, XYZ Corp. Employee Info Server, etc. As the example attempts to illustrate, some service classes are well known while others are unique and specific to a particular vertical application. In either case, every service class is represented by both a class name and a class identifier. The class name does not necessarily need to be unique, but the class identifier must be. Globally Unique Identifiers (GUIDs) are used to represent service class identifiers. For well-known services, class names and class identifiers (GUIDs) have been preallocated, and macros are available to convert between, for example, TCP port numbers (in host-byte order) and the corresponding class identifier GUIDs. For other services, the developer chooses the class name and uses the Uuidgen.exe utility to generate a GUID for the class identifier.

The notion of a service class exists to allow a set of attributes to be established that are held in common by all instances of a particular service. This set of attributes is provided at the time the service class is defined to Winsock, and is referred to as the service class schema information. When a service is installed and made available on a host computer, 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 namespace.

Note that the installation of a service class only needs to occur on computers where the service executes, not on all of the clients which may utilize the service. Where possible, the Ws2\_32.dll provides service class schema information to a namespace 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 namespace provider that has indicated its ability to supply this data. Since there is no guarantee that the Ws2\_32.dll can supply the service class schema, namespace providers that need this information must have a fallback mechanism to obtain it through namespace-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 preallocated (**SVCID\_HOSTNAME**). A query that specifies **SVCID\_HOSTNAME** as the service class and specifies a host name for the service instance name will return host address information if the query is successful.

In Windows Sockets 2, applications that are protocol-independent should 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 over TCP. In this case, a successful query returns a complete transport address for the specified service on the indicated host, and the application is not required to

inspect the internals of a [sockaddr](#) structure.

The Internet's Domain Name System does not have a well-defined means to store service class schema information. As a result, DNS namespace providers for Winsock can only 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 HTTP, 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 "ftp.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 computer 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.

## Related topics

[DNS Reference](#)

[Name Resolution Data Structures](#)

[Protocol-Independent Name Resolution](#)

[Registration and Name Resolution](#)

[SOCKADDR](#)

[Summary of Name Resolution Functions](#)

# Summary of Name Resolution Functions

3/5/2021 • 3 minutes to read • [Edit Online](#)

The name resolution functions can be grouped into three categories: Service installation, client queries, and helper (with 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 identifier, and a series of [WSANSCLASSINFO](#) structures. These structures are each specific to a particular namespace, 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 identifier.

Once a service class exists, specific instances of a service can be installed or removed through [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 identifier, service name (for this instance), applicable namespace identifier and protocol information, and a set of transport addresses at which the service listens. Services should invoke [WSASetService](#) when they initialize to advertise their presence in dynamic namespaces.

## Client Query

- [WSAEnumNameSpaceProviders](#)
- [WSALookupServiceBegin](#)
- [WSALookupServiceNext](#)
- [WSALookupServiceEnd](#)

The [WSAEnumNameSpaceProviders](#) function allows an application to discover which namespaces are accessible through Winsock name resolution facilities. It also allows an application to determine whether a given namespace is supported by more than one namespace provider, and to discover the provider identifier for any particular namespace provider. Using a provider identifier, the application can restrict a query operation to a specified namespace provider.

Winsock namespace-query operations involve 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 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 Sockets 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 2, however, applications should check for both `WSAENOMORE` and `WSA_E_NO_MORE` for the widest possible compatibility with namespace 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 identifier, a pair of functions used to translate a transport address between a `SOCKADDR` structure 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 preallocated GUIDs.

The following macros from Winsock2.h aid in mapping between well known service classes and these namespaces:

MACRO	DESCRIPTION
SVCID_TCP(Port) SVCID_UDP(Port) SVCID_NETWARE(Object Type)	Given a port for TCP/IP or UDP/IP or the object type in the case of NetWare, returns the GUID (port number in host order).
IS_SVCID_TCP(GUID)IS_SVCID_UDP(GUID) IS_SVCID_NETWARE(GUID)	Returns TRUE if the GUID is within the allowable range.
SET_TCP_SVCID(GUID, port)SET_UDP_SVCID(GUID, port)	Initializes a GUID structure with the GUID equivalent for a TCP or UDP port number (port number must be in host order).
PORTRFROM_SVCID_TCP(GUID)PORTRFROM_SVCID_UDP(GUID) SAPIDFROM_SVCID_NETWARE(GUID)	Returns the port or object type associated with the GUID (port number in host order).

## Related topics

[getaddrinfo](#)

[GetAddrInfoEx](#)

[GetAddrInfoW](#)

[getnameinfo](#)

[GetNameInfoW](#)

[Name Resolution Data Structures](#)

[Name Resolution Model](#)

[Protocol-Independent Name Resolution](#)

Registration and Name Resolution

**SOCKADDR**

**WSAEnumNameSpaceProviders**

**WSAGetServiceClassNameByClassId**

**WSAInstallServiceClass**

**WSALookupServiceBegin**

**WSALookupServiceEnd**

**WSALookupServiceNext**

**WSARemoveServiceClass**

**WSASetService**

**WSAQUERYSET**

**WSANSCLASSINFO**

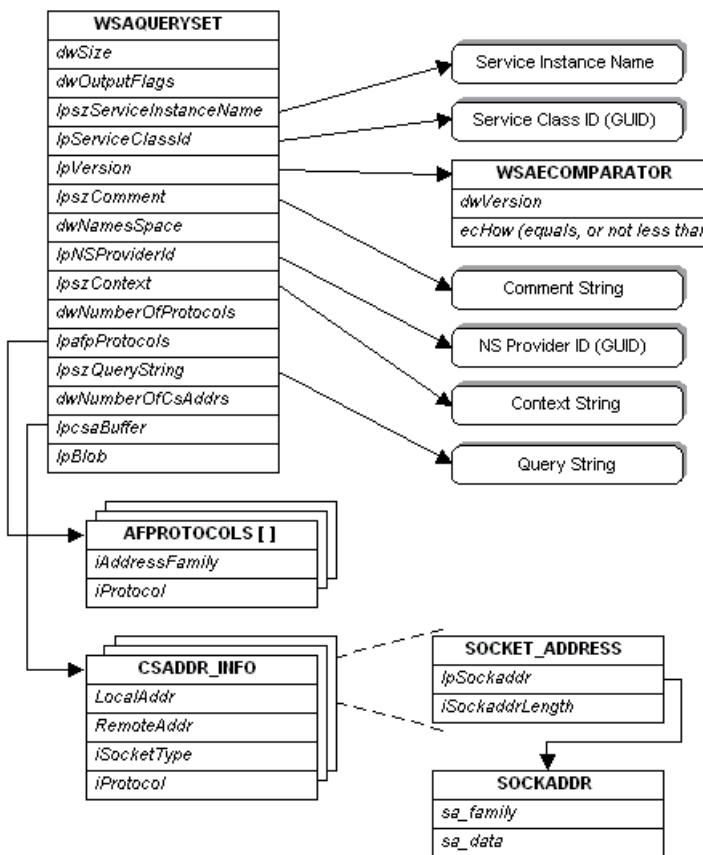
# Name Resolution Data Structures

3/5/2021 • 4 minutes to read • [Edit Online](#)

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

## 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 the [WSAQUERYSET](#) structure and the structures it references is illustrated as follows.



Within the [WSAQUERYSET](#) structure, most of the member are self explanatory, but some deserve additional explanation. The `dwSize` member must always be filled in with `sizeof(WSAQUERYSET)`, as this is used by namespace providers to detect and adapt to different versions of the [WSAQUERYSET](#) structure that may appear over time.

The `dwOutputFlags` member is used by a namespace provider to provide additional information about query results. For details, see the [WSALookupServiceNext](#) function.

The [WSAECOMPATOR](#) structure referenced by the `lpVersion` member is used for both query constraint and results. For queries, the `dwVersion` member indicates the desired version of the service. The `ecHow` member is an enumerated type which specifies how the comparison can 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 the `dwVersion` member.

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 **IpszContext** member applies to hierarchical namespaces, 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 ("") starts the search at the default context.
- A value of "\ starts the search at the top of the namespace.
- 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 namespace specific. If the **dwNameSpace** member is **NS\_ALL**, then only "" or "\" should be passed as the context since these are recognized by all namespaces.

The **IpszQueryString** member is used to supply additional, namespace-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 the **IpafpProtocols** member 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.

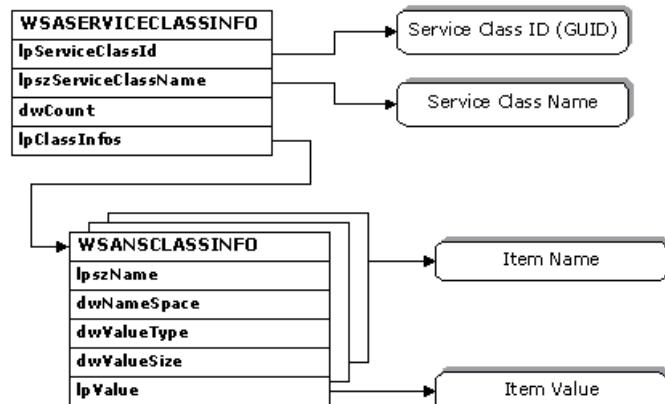
The array of the **CSADDR\_INFO** structure referenced by the **IpcsaBuffer** member contain all of the information needed for either a service to use in establishing a listen, or for a client to use in establishing a connection to the service. The **LocalAddr** and **RemoteAddr** members both directly contain a **SOCKET\_ADDRESS** structure.

A service would create a socket by calling the **socket** or **WSASocket** function using the tuple of *LocalAddr:lpSockaddr->sa\_family*, *iSocketType*, and *iProtocol* as parameters. A service would bind the socket to a local address by calling the **bind** function using *LocalAddr:lpSockaddr* and *LocalAddr:lpSockaddrLength* as parameters.

A client creates its socket by calling the **socket** or **WSASocket** function using the tuple of *LocalAddr:lpSockaddr->sa\_family*, *iSocketType*, and *iProtocol* as parameters. A client uses the combination of *RemoteAddr:lpSockaddr* and *RemoteAddr:lpSockaddrLength* as parameters when making a remote connection using the **connect**, **ConnectEx**, or **WSAConnect** function.

## 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 that contain a series of members that apply to specific namespaces. A class info data structure is as follows:



For each service class, there is a single **WSASERVICECLASSINFO** structure. Within the **WSASERVICECLASSINFO** structure, the service class' unique identifier is contained in the **IpServiceClassId** member, and an associated display string is referenced by the **IpszServiceClassName** member. This is the string that is returned by the **WSAGetServiceClassNameByClassId** function.

The **IpClassInfos** member in the [WSASERVICECLASSINFO](#) structure references an array of [WSANSCLASSINFO](#) structures, each of which supplies a named and typed member that applies to a specified namespace. Examples of values for the **IpszName** member include: "SapId", "TcpPort", "UdpPort", etc. These strings are generally specific to the namespace identified in the **dwNameSpace** member. Typical values for the **dwValueType** member might be REG\_DWORD, REG\_SZ, etc. The **dwValueSize** member indicates the length of the data item pointed to by **IpValue**.

The entire collection of data represented in a [WSASERVICECLASSINFO](#) structure is provided to each namespace provider when the [WSAInstallServiceClass](#) function is invoked. Each individual namespace provider then sifts through the list of [WSANSCLASSINFO](#) structures and retains the information applicable to it.

## Related topics

[AFPROTOCOLS](#)

[CSADDR\\_INFO](#)

[Name Resolution Model](#)

[Protocol-Independent Name Resolution](#)

[Registration and Name Resolution](#)

[SOCKET\\_ADDRESS](#)

[Summary of Name Resolution Functions](#)

[WSAECOMPARATOR](#)

[WSAGetServiceClassNameByClassId](#)

[WSAInstallServiceClass](#)

[WSALookupServiceBegin](#)

[WSAQUERYSET](#)

[WSASERVICECLASSINFO](#)

# Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 API

3/5/2021 • 2 minutes to read • [Edit Online](#)

## NOTE

All of the Windows Sockets 1.1 functions for name resolution are specific to IPv4 TCP/IP networks. Application developers are strongly discouraged from continuing to utilize these transport-specific functions that only support IPv4.

Application developers should be using the following functions that are protocol-independent and support both IPv6 and IPv4 name resolution.

- [getaddrinfo](#)
- [GetAddrInfoEx](#)
- [GetAddrInfoW](#)
- [getnameinfo](#)
- [GetNameInfoW](#)

Windows Sockets 1.1 defined a number of routines used for name resolution with TCP/IP (IP version 4) networks. These are sometimes called the **getXbyY** functions and include the following:

[gethostname](#)  
[gethostbyaddr](#)  
[gethostbyname](#)  
[getprotobynumber](#)  
[getprotobynumber](#)  
[getservbyname](#)  
[getservbyport](#)

Asynchronous versions of these functions were also defined.

[WSAAAsyncGetHostByAddr](#)  
[WSAAAsyncGetHostByName](#)  
[WSAAAsyncGetProtoByName](#)  
[WSAAAsyncGetProtoByNumber](#)  
[WSAAAsyncGetServByName](#)  
[WSAAAsyncGetServByPort](#)

There are also two functions, now implemented in the Winsock2.dll, used to convert dotted IPv4 address notation to and from string and binary representations, respectively.

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

In order to retain strict backward compatibility with Windows Sockets 1.1, all of the older IPv4-only functions continue to be supported as long as at least one namespace 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 WSs2\_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.

This section describes compatible name resolution for TCP/IP in the Windows Sockets 1.1 API. The following list describes the topics in this section:

- [Basic Approach for GetXbyY in the API](#)
- [getprotobynumber and getprotobyname Functions in the API](#)
- [getservbyport and getservbyname Functions in the API](#)
- [gethostbyname Function in the API](#)
- [gethostbyaddr Function in the API](#)
- [gethostname Function in the API](#)

## Related topics

[Protocol-Independent Name Resolution](#)

[Registration and Name Resolution](#)

# Basic Approach for GetXbyY in the API

3/5/2021 • 2 minutes to read • [Edit Online](#)

Most `getXbyY` functions are translated by the `Ws2_32.dll` to a `WSALookupServiceBegin`, `WSALookupServiceNext`, and `WSALookupServiceEnd` 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 name service providers that support `AF_INET`. Whenever a `GetXbyY` function returns a `HOSTENT` or `SERVENT` structure, the `Ws2_32.dll` specifies the `LUP_RETURN_BLOB` flag in `WSALookupServiceBegin` so that the desired information is returned by the name service provider. 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 parameters must, of course, be completely contained within the blob, and all strings are ASCII.

## Related topics

[Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 API](#)

[Protocol-Independent Name Resolution](#)

[Registration and Name Resolution](#)

# getprotobynumber and getprotobyname Functions in the API

3/5/2021 • 2 minutes to read • [Edit Online](#)

The [getprotobynumber](#) and [getprotobynumber](#) functions are implemented within the Ws2\_32.dll by consulting a local protocols database. They do not result in any name resolution query.

## Related topics

[Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 API](#)

[Protocol-Independent Name Resolution](#)

[Registration and Name Resolution](#)

# getservbyname and getservbyport Functions in the API

3/5/2021 • 2 minutes to read • [Edit Online](#)

The [getservbyname](#) and [getservbyport](#) functions use the [WSALookupServiceBegin](#) function to query `SVCID_INET_SERVICEBYNAME` as the service class GUID. The `/pszServiceInstanceName` member in the [WSAQUERYSET](#) structure passed to the [WSALookupServiceBegin](#) function references a string to indicate the service name or service port, and (optionally) the service protocol. The formatting of the string is illustrated as FTP or 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 namespace provider will place a [SERVENT](#) structure in the blob (using offsets instead of pointers as described above). Namespace providers should honor these other `LUP_RETURN_*` flags as well.

FLAG	DESCRIPTION
<code>LUP_RETURN_NAME</code>	Returns the <code>s_name</code> member from <a href="#">SERVENT</a> structure in <code>/pszServiceInstanceName</code> .
<code>LUP_RETURN_TYPE</code>	Returns canonical GUID in <code>/pServiceClassId</code> . It is understood that a service identified as FTP or 21 may be on another port according to locally established conventions. The <code>s_port</code> parameter of the <a href="#">SERVENT</a> structure should indicate where the service can be contacted in the local environment. The canonical GUID returned when <code>LUP_RETURN_TYPE</code> is set should be one of the predefined GUIDs from <code>Svcs.h</code> that corresponds to the port number indicated in the <a href="#">SERVENT</a> structure.

## Related topics

[Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 API](#)

[Protocol-Independent Name Resolution](#)

[Registration and Name Resolution](#)

# gethostbyname Function in the API

3/5/2021 • 2 minutes to read • [Edit Online](#)

The [gethostbyname](#) function uses the [WSALookupServiceBegin](#) function to query `SVCID_INET_HOSTADDRBYNAME` as the service class GUID. The host name is supplied in the `IpszServiceInstanceName` member in the [WSAQUERYSET](#) structure passed to the [WSALookupServiceBegin](#) function. The `Ws2_32.dll` specifies `LUP_RETURN_BLOB` and the name service provider places a [HOSTENT](#) structure in the blob (using offsets instead of pointers as described above). Name service providers should honor these other `LUP_RETURN_*` flags as well.

FLAG	DESCRIPTION
<code>LUP_RETURN_NAME</code>	Returns the <code>h_name</code> member from the <a href="#">HOSTENT</a> structure in <code>IpszServiceInstanceName</code> .
<code>LUP_RETURN_ADDR</code>	Returns addressing information from <a href="#">HOSTENT</a> in <a href="#">CSADDR_INFO</a> structures, port information is defaulted to zero. Note that this routine does not resolve host names that consist of a dotted IPv4 address.

## Related topics

[Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 API](#)

[Protocol-Independent Name Resolution](#)

[Registration and Name Resolution](#)

# gethostbyaddr Function in the API

3/5/2021 • 2 minutes to read • [Edit Online](#)

The [gethostbyaddr](#) function uses the [WSALookupServiceBegin](#) function to query `SVCID_INET_HOSTNAMEBYADDR` as the service class GUID. The host address is supplied as a dotted decimal IPv4 string (for example, "192.9.200.120") in the `lpszServiceInstanceName` member of the [WSAQUERYSET](#) structure passed to the [WSALookupServiceBegin](#) function. The `Ws2_32.dll` specifies `LUP_RETURN_BLOB` and the name service provider places a [HOSTENT](#) structure in the blob (using offsets instead of pointers as described above). Name service providers should honor these other `LUP_RETURN_*` flags as well.

FLAG	DESCRIPTION
<code>LUP_RETURN_NAME</code>	Returns the <code>h_name</code> member from <a href="#">HOSTENT</a> structure in <code>lpszServiceInstanceName</code> .
<code>LUP_RETURN_ADDR</code>	Returns addressing information from <a href="#">HOSTENT</a> in <a href="#">CSADDR_INFO</a> structures, port information is defaulted to zero.

## Related topics

[Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 API](#)

[Protocol-Independent Name Resolution](#)

[Registration and Name Resolution](#)

# gethostname Function in the API

3/5/2021 • 2 minutes to read • [Edit Online](#)

The [gethostname](#) function uses the [WSALookupServiceBegin](#) function to query SVCID\_HOSTNAME as the service class GUID. If the `IpszServiceInstanceName` member of the [WSAQUERYSET](#) structure passed to the [WSALookupServiceBegin](#) function is `NULL` or references a `NULL` string (that is . ""), the local host is to be resolved. Otherwise, a lookup on a specified host name occurs. For the purposes of emulating [gethostname](#) the Ws2\_32.dll specifies a `NULL` pointer for the `IpszServiceInstanceName` member, and specifies `LUP_RETURN_NAME` so that the host name is returned in the `IpszServiceInstanceName` member. If an application uses this query and specifies `LUP_RETURN_ADDR` then the host address is provided in a [CSADDR\\_INFO](#) structure. The `LUP_RETURN_BLOB` action is undefined for this query. Port information is defaulted to zero unless the `IpszQueryString` member of the [WSAQUERYSET](#) structure passed to the [WSALookupServiceBegin](#) function references a service such as FTP, in which case the complete transport address of the indicated service is supplied.

## Related topics

[Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 API](#)

[Protocol-Independent Name Resolution](#)

[Registration and Name Resolution](#)

# Multipoint and Multicast Semantics

3/5/2021 • 2 minutes to read • [Edit Online](#)

In considering how to support multipoint and multicast semantics in Windows Sockets 2 (Winsock), a number of existing and proposed schemes (including IP-multicast, ATM point-to-multipoint connection, ST-II, T.120, H.320-MCU) were examined. To enable a coherent discussion of the various schemes, it is valuable to first create a taxonomy that characterizes the essential attributes of each. In this document, the term *multipoint* represents both multipoint and multicast.

# Multipoint Taxonomy

3/5/2021 • 3 minutes to read • [Edit Online](#)

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 among session participants.

## Session Establishment in the Control Plane

In the control plane there are two distinct types of session establishment:

- rooted
- nonrooted

In the case of rooted control, a special participant, `c_root`, exists that is different from the rest of the members of this multipoint session, each of which are called a `c_leaf`. The `c_root` must remain present for the whole duration of the multipoint session, as the session is 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 nonrooted control plane, all the members belonging to a multipoint session are leaves, that is, no special participant acting as a `c_root` exists. Each `c_leaf` must add itself to a preexisting multipoint session that is always available (as in the case of an IP multicast address), or has been set up through some out-of-band (OOB) mechanism that 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 nonrooted 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 an H.320 video conference.

## Data Transfer in the Data Plane

In the data plane, there are two types of data transfer styles:

- rooted
- nonrooted

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 are referred to as a `d_leaf`. The traffic could be unidirectional or bidirectional. The data sent out from the `d_root` is duplicated (if required) and delivered to every `d_leaf`, while the data from `d_leafs` only goes to the `d_root`. In the case of a rooted data plane, no traffic is allowed among `d_leafs`. An example of a protocol that is rooted in the data plane is ST-II.

In a nonrooted data plane, all the participants are equal, that is, any data they send is delivered to all the other participants in the same multipoint session. Likewise each `d_leaf` node can receive data from all other `d_leafs`, and in some cases, from other nodes that are not participating in the multipoint session. No special `d_root` node exists. IP-multicast is nonrooted 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 irrelevant to the interface the application would use to perform multipoint communications. Therefore these issues are not addressed in this appendix or 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 do not appear to be any existing schemes that employ a nonrooted control plane along with a rooted data plane.

	ROOTED CONTROL PLANE	NONROOTED (IMPLICIT ROOTED) CONTROL PLANE
Rooted data plane	ATM, ST-II	No known examples.
Nonrooted data plane	T.120	IP-multicast, H.320 (MCU)

# Windows Sockets 2 Interface Elements for Multipoint and Multicast

3/5/2021 • 2 minutes to read • [Edit Online](#)

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](#) structure.
- Four flags defined for the *dwFlags* parameter of [WSASocket](#).
- One function, [WSAJoinLeaf](#), for adding leaf nodes into a multipoint session.
- Two [WSAioctl](#) command codes for controlling multipoint loopback and the scope of multicast transmissions.

The following sections describe these interface elements in more detail:

- [Semantics for Joining Multipoint Leaves](#)
- [How Existing Multipoint Protocols Support These Extensions](#)

# Attributes in WSAPROTOCOL\_INFO Structure

3/5/2021 • 2 minutes to read • [Edit Online](#)

In support of the taxonomy described above, three attribute parameters in the [WSAPROTOCOL\\_INFO](#) structure are used to distinguish the schemes used in the control and data planes respectively:

- XP1\_SUPPORT\_MULTIPOINT with a value of 1 indicates this protocol entry supports multipoint communications, and that the following two parameters are meaningful.
- XP1\_MULTIPOINT\_CONTROL\_PLANE indicates whether the control plane is rooted (value = 1) or nonrooted (value = 0).
- XP1\_MULTIPOINT\_DATA\_PLANE indicates whether the data plane is rooted (value = 1) or nonrooted (value = 0).

Note that two [WSAPROTOCOL\\_INFO](#) entries would be present if a multipoint protocol supported both rooted and nonrooted 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

3/5/2021 • 2 minutes to read • [Edit Online](#)

In some instances sockets joined to a multipoint session may have some differences in behavior 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 indicate 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"
- "c\_leaf /d\_leaf"

# SIO\_MULTIPOINT\_LOOPBACK Command Code for WSALoctl

3/5/2021 • 2 minutes to read • [Edit Online](#)

When d\_leaf sockets are used in a nonrooted data plane, it is desirable to have traffic that is sent out received back on the same socket. The SIO\_MULTIPOINT\_LOOPBACK command code for [WSALoctl](#) is used to enable or disable loopback of multipoint traffic.

# SIO\_MULTICAST\_SCOPE Command Code for WSALoctl

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 1 (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 the following for a discussion on how this function is utilized.

# Semantics for Joining Multipoint Leaves

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 to 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 receives an FD\_ACCEPT indication when invited to join the session, and signals its willingness to join by calling **WSAAccept**. The root application then receives 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 nonrooted 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 usable 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 sometimes 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 this 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** acting as a:

- Multipoint root and inviting a new leaf to join the session
- Leaf making an admittance request to a rooted multipoint session
- Leaf seeking admittance to a nonrooted multipoint session (for example, IP multicast)

# Using WSAJoinLeaf

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 **WSACConnect** 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 nonblocking mode, the returned socket descriptor is not usable until after a corresponding FD\_CONNECT indication is 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** differs 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 the exchange of multipoint data, but rather it is used to receive FD\_XXX indications (for example, FD\_CLOSE) for the connection that exists to the particular c\_leaf. Some multipoint implementations may also allow this socket to be used for side chats between the root and an individual leaf node. An FD\_CLOSE indication is 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** causes the socket in the corresponding leaf node to get an 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 a rooted control scheme, the root application puts its c\_root socket in listening mode by calling **listen**. The standard FD\_ACCEPT notification is 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

3/5/2021 • 2 minutes to read • [Edit Online](#)

In the control plane, there are some significant semantic differences between a c\_root socket and a regular point-to-point socket:

- The c\_root socket can be used in [WSAJoinLeaf](#) to join a new a leaf.
- Placing a c\_root socket into listening mode (by calling [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.
- The closing of a c\_root socket causes all of the associated c\_leaf sockets to get FD\_CLOSE notification.

There are 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:

- The data sent on the d\_root socket is delivered to all the leaves in the same multipoint session.
- 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 nonrooted data plane, the data sent on the d\_leaf socket goes 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 nonrooted data plane is contained in the corresponding [WSAPROTOCOL\\_INFO](#) structure for the socket.

# How multipoint protocols support these extensions

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following topics indicate how IP multicast and ATM point-to-multipoint capabilities can be accessed through the Windows Sockets 2 multipoint functions. We chose these two as examples because they are popular and well understood.

- [IP Multicast](#)
- [ATM Point to Multipoint](#)

# IP Multicast

3/5/2021 • 2 minutes to read • [Edit Online](#)

IP multicast falls into the category of nonrooted data plane and nonrooted 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 Internet Engineering Task Force (IETF). Five operations are thus made available:

- `IP_MULTICAST_TTL`—Sets time-to-live, controls scope of a multicast session.
- `IP_MULTICAST_IF`—Determines interface to be used for multicasting.
- `IP_ADD_MEMBERSHI`—Joins a specified multicast session.
- `IP_DROP_MEMBERSHIP`—Drops out of a multicast session.
- `IP_MULTICAST_LOOP`—Controls 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 through a TCP/IP-specific socket option as described in the Windows Sockets 2 Protocol-Specific Annex. The remaining three operations are covered well with the Windows Sockets 2 semantics described here.

The application would open sockets with `c_leaf/d_leaf` flags in [WSASocket](#). It would use [WSAJoinLeaf](#) to add itself to a multicast group on the default interface designated for multicast operations. If the flag in [WSAJoinLeaf](#) indicates that this socket is only a sender, then the join operation is essentially a no-op and no IGMP messages need to be sent. Otherwise, an IGMP packet is sent out to the router to indicate interests in receiving packets sent to the specified multicast address. Since the application created special `c_leaf/d_leaf` sockets used only for performing multicast, the standard [closesocket](#) function would be used to drop out of the multicast session. The `SIO_MULTIPOINT_LOOPBACK` command code for [WSAIoctl](#) provides a generic control mechanism for determining whether data sent on a `d_leaf` socket in a nonrooted multipoint scheme can also be received on the same socket.

## NOTE

The Winsock version of the `IP_MULTICAST_LOOP` option is semantically different than the UNIX version of the `IP_MULTICAST_LOOP` option:

- In Winsock, the `IP_MULTICAST_LOOP` option applies only to the receive path.
- In the UNIX version, the `IP_MULTICAST_LOOP` option applies to the send path.

For example, applications ON and OFF (which are easier to track than X and Y) join the same group on the same interface; application ON sets the `IP_MULTICAST_LOOP` option on, application OFF sets the `IP_MULTICAST_LOOP` option off. If ON and OFF are Winsock applications, OFF can send to ON, but ON cannot send to OFF. In contrast, if ON and OFF are UNIX applications, ON can send to OFF, but OFF cannot send to ON.

# ATM Point to Multipoint

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 uses **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 is mapped to the Q.2931 ADDPARTY message. The leaf-initiated join is not supported in ATM UNI 3.1, but is supported in ATM UNI 4.0. Thus **WSAJoinLeaf** with a c\_leaf socket specified is mapped to the appropriate ATM UNI 4.0 message.

There are additional considerations to bear in mind for ATM point-to-multipoint:

- The addition of new leaves to the ATM point-to-multipoint session is invitation-based only. The root invites leaves — which have already their **accept** function call — by calling the **WSAJoinLeaf** function.
- The flow of data in an ATM point-to-multipoint session is from root-to-leaves only; leaves cannot use the same session to send information to the root.
- Only one leaf per ATM adapter is allowed.

# Internet Protocol Version 6 (IPv6)

3/5/2021 • 2 minutes to read • [Edit Online](#)

It is important to ensure that new Winsock applications as well as existing applications are fully compatible with IPv6. The availability of the IPv4 address space for new IPv4 address allocations was exhausted for Asia and the Pacific in 2011. Other parts of the world are expected to be exhausted in a few years.

A growing percentage of new websites and services are available using IPv6 addresses. Many Internet users in emerging markets are dependent on IPv6 for Internet access.

Microsoft has a long commitment of supporting IPv6. Full IPv6 support is provided on Windows XP with Service Pack 1 (SP1) and later.

This document provides information about the Winsock support for IPv6:

- [Using IPv6](#)
- [Using Internet Explorer to Access IPv6 Websites](#)
- [Recommended Configurations for IPv6](#)
- [Additional IPv6 Topics](#)

For additional information on adding IPv6 capability to your Windows Sockets applications, see [IPv6 Guide for Windows Sockets Applications](#).

An introduction to the IPv6 protocol along with overviews on deployment and IPv6 transitioning technologies is available on Technet at [Microsoft Internet Protocol Version 6 \(IPv6\)](#).

## Related topics

[IPv6 Guide for Windows Sockets Applications](#)

[IPv6 Support](#)

[IPv6 Technology Preview for Windows 2000](#)

[Microsoft Internet Protocol Version 6 \(IPv6\)](#)

# Using IPv6

3/5/2021 • 2 minutes to read • [Edit Online](#)

On Windows XP and later, a new command-line tool is provided for configuring and managing IPv4. This tool uses the "netsh interface ip" command for configuring and managing IPv4.

On Windows XP with Service Pack 1 (SP1) and later, this new command-line tool was enhanced to support the configuration and management of IPv6. This enhanced tool is the "netsh interface ipv6" command.

Configuration changes made using the Netsh.exe commands are permanent and are not lost when the computer or the IPv6 protocol is restarted.

The following command is available on Windows XP with SP1 and later to query and configure IPv6 on a local computer:

- [Netsh.exe](#)

Prior to Windows XP with Service Pack 1 (SP1), IPv6 configuration and management used several older command-line tools to configure and manage IPv6. Using these older tools, the IPv6 changes are not permanent and are lost when the computer or the IPv6 protocol was restarted.

The following older commands are available on Windows XP

- [Net.exe](#)
- [Ipv6.exe](#)
- [Ipsec6.exe](#)

These older tools were also provided in the IPv6 Technology Preview for Windows 2000

Configuration changes using these older tools could be maintained by placing them as command lines in a command script file (.cmd) that is run after restarting the computer or the IPv6 protocol. To reinstate configuration changes automatically after restarting, it was possible to use the Windows Scheduled Tasks to run the .cmd file when the computer starts.

These older tools are not provided on Windows Server 2003 and later. The "netsh interface ipv6" tool is provided for configuring and managing IPv6 from the command-line on Windows Server 2003 and later.

## Related topics

[Internet Protocol Version 6 \(IPv6\)](#)

[IPv6 Guide for Windows Sockets Applications](#)

[IPv6 Technology Preview for Windows 2000](#)

# Netsh.exe

3/5/2021 • 2 minutes to read • [Edit Online](#)

The Netsh commands for IPv6 provide a command-line tool that you can use to query and configure IPv6 interfaces, address, caches, and routes. The Netsh interface IPv6 commands are supported on Windows XP with Service Pack 1 (SP1) and later.

Netsh.exe has many subcommands for IPv6. A complete list of options for Netsh Interface IPv6 is available from the command prompt on Windows XP with SP1 and later by typing the following:

**netsh interface ipv6 /?**

Documentation on all of the **netsh** commands for IPv6 is also available online on Technet. For more information on **netsh** on Windows Server 2008, please see [Netsh commands for Interface \(IPv4 and IPv6\)](#). For more information on **netsh** on Windows Server 2003, please see [Netsh commands for Interface IPv6](#).

The following are a few commonly used commands for IPv6, although many other commands are supported:

**netsh interface ipv6 add address**

Adds an IPv6 address to a specific interface on the local computer. This command has suboption parameters that must to be specified.

**netsh interface ipv6 add dns**

Adds a DNS server IPv6 address to the statically-configured list of DNS servers for the specified interface on the local computer. This command has suboption parameters that must to be specified.

**netsh interface ipv6 add route**

Adds a route for a specified IPv6 prefix address on the local computer. This command has suboption parameters that must to be specified.

**netsh interface ipv6 delete address**

Deletes the specified IPv6 address from a specific interface on the local computer. This command has suboption parameters that must to be specified.

**netsh interface ipv6 delete dns**

Deletes a DNS server address from the statically-configured list of DNS servers for the specified interface on the local computer. This command has suboption parameters that must to be specified.

**netsh interface ipv6 delete interface**

Deletes a specified interface from the IPv6 stack on the local computer. This command has suboption parameters that must to be specified.

**netsh interface ipv6 delete route**

Deletes a route for a specified IPv6 prefix address on the local computer. This command has suboption parameters that must to be specified.

**netsh interface ipv6 dump**

Creates a script that contains the commands to create the current configuration. If saved to a file, this script can be used to restore altered configuration settings.

**netsh interface ipv6 install**

Installs the IPv6 protocol on the local computer.

**netsh interface ipv6 renew**

Restarts the IPv6 interfaces on the local computer.

**netsh interface ipv6 reset**

Resets the IPv6 configuration state on the local computer.

**netsh interface ipv6 show global**

Displays the global configuration parameters for IPv6 on the local computer.

**netsh interface ipv6 show address**

Displays all IPv6 addresses or all IPv6 addresses on a specific interface on the local computer. This command has suboption parameters that may need to be specified.

**netsh interface ipv6 uninstall**

Uninstalls the IPv6 protocol on the local computer.

## Netsh commands for IPv4

Similar Netsh commands are available for IPv4. A complete list of options for Netsh commands for use with IPv4 is available from the command prompt on Windows XP with SP1 and later by typing the following:

**netsh interface ip /?**

Documentation on all of the Netsh commands for IPv4 is also available online on Technet. For more information, please see [Netsh commands for Interface IP](#)

# Net.exe

3/5/2021 • 2 minutes to read • [Edit Online](#)

Net.exe can be used to stop and start the IPv6 protocol. Restarting IPv6 reinitializes the protocol as if the computer were restarting, which may change interface numbers. Net.exe has many subcommands. The following commands are relevant to IPv6:

**net stop tcpip6**

Stops the IPv6 protocol and unloads it from memory. This command fails if any IPv6 sockets are open.

**net start tcpip6**

Starts the IPv6 protocol if it was stopped. If a new Tcpip6.sys driver file is present in the %systemroot%\System32\Drivers directory, that driver file is loaded.

# Ipv6.exe

3/5/2021 • 5 minutes to read • [Edit Online](#)

All IPv6 configuration is done with the Ipv6.exe tool. This tool is primarily used for the querying and configuring of IPv6 interfaces, addresses, caches, and routes. The following are IPv6 subcommands:

## **ipv6 if [if#]**

Displays information about interfaces. If an interface number is specified, only information about that interface is displayed. Otherwise, information about all interfaces is displayed. The output includes the interface's link-layer address and the list of IPv6 addresses assigned to the interface, including the interface's current MTU and the maximum (true) MTU that the interface can support.

Interface #1 is a pseudo-interface used for loopback. Interface #2 is a pseudo-interface used for configured tunneling, automatic tunneling, and 6to4 tunneling. Other interfaces are numbered sequentially in the order in which they are created. This order varies from one computer to another.

Link-layer addresses in *aa-bb-cc-dd-ee-ff* format are Ethernet addresses. Link-layer address in *a.b.c.d* form are 6-over-4 interfaces. For more information on 6-over-4, see RFC 2529.

The two pseudo-interfaces do not use IPv6 Neighbor Discovery.

## **ipv6 ifc if# [forwards] [advertisers] [-forwards] [-advertisers] [mtu #bytes] [site site-identifier]**

Controls interface attributes. Interfaces can be forwarding, in which case they forward packets with destination addresses not assigned to the interface. Interfaces can be advertising, in which case they send router advertisements. These attributes can be independently controlled. An interface either sends router solicitations and receives router advertisements, or receives router solicitations and sends router advertisements.

Because the two pseudo-interfaces do not use Neighbor Discovery, they cannot be configured to send router advertisements.

Forwards can be abbreviated as forw, and advertised as adv.

The MTU for the interface can also be set. The new MTU must be less than or equal to the link's maximum (true) MTU (as reported by ipv6 if) and greater than or equal to the minimum IPv6 MTU (1280 bytes).

The site-identifier for an interface can also be changed. Site identifiers are used in the sin6\_scope\_id field with site-local addresses.

## **ipv6 ifd if#**

Deletes an interface. The loopback and tunnel pseudo-interfaces cannot be deleted.

## **ipv6 nc [if# [address]]**

Displays the contents of the neighbor cache. If an interface number is specified, only the contents of that interface's neighbor cache are displayed. Otherwise, the contents of all the interface's neighbor caches are displayed. If an interface is specified, an IPv6 address can be specified to display only that neighbor cache entry.

For each neighbor cache entry, the interface, IPv6 address, link-layer address, and reach state are displayed.

## **ipv6 ncf [if# [address]]**

Flushes the specified neighbor cache entries. Only neighbor cache entries without references are purged. Because route cache entries hold references to neighbor cache entries, the route cache should be flushed first.

Routing table entries can also hold references to neighbor cache entries.

#### **ipv6 rc [if# address]**

Displays the contents of the route cache. The route cache is the Microsoft IPv6 implementation name for the destination cache. If an interface and address are specified, the route cache entry for reaching the address through the interface is displayed. Otherwise, all route cache entries are displayed.

For each route cache entry, the IPv6 address and the current next-hop interface and neighbor address are displayed. The preferred source address for use with this destination, the current path MTU for reaching this destination through the interface, and the determination of whether this is an interface specific–route cache entry are also displayed. Any care-of address (for mobility) for the destination address is displayed as well.

A destination address can have multiple route cache entries—as many as one for each outgoing interface. However, a destination address can have a maximum of one route cache entry that is not interface-specific. An interface specific–route cache entry is only used if the application explicitly specifies that outgoing interface.

#### **ipv6 rcf [if# [address]]**

Flushes the specified route cache entries.

#### **ipv6 bc**

Displays the contents of the binding cache, which holds bindings between home addresses and care-of addresses for mobile IPv6.

For each binding, the home address, care-of address, binding sequence number, and lifetime are displayed.

#### **ipv6 adu if#/address [lifetime VL[/PL]] [anycast] [unicast]**

Adds or removes a unicast or anycast address assignment on an interface. The unicast address is acted upon unless anycast is specified.

Lifetime is infinite if unspecified. If only a valid lifetime is specified, the preferred lifetime is equal to the valid lifetime. An infinite lifetime may be specified, or a finite value in seconds. The preferred lifetime must be less than or equal to the valid lifetime. Specifying a lifetime of zero causes the address to be removed.

You can abbreviate lifetime as life.

For anycast addresses, the only valid lifetime values are zero and infinite.

#### **ipv6 spt**

Displays the current contents of the site prefix table.

For each site prefix, the command displays the prefix, the interface to which the site prefix applies, and the prefix lifetime in seconds.

Site prefixes are normally auto-configured from router advertisements. They are used in the [getaddrinfo](#) function to filter inappropriate site-local addresses.

#### **ipv6 spu prefix if# [lifetime L]**

Adds, removes, or updates a prefix in the site prefix table.

The prefix and interface number are required. The site prefix lifetime, specified in seconds, defaults to infinite if not specified. Specifying a lifetime of zero deletes the site prefix.

This command is unnecessary for the normal configuration of hosts or routers.

#### **ipv6 rt**

Displays the current contents of the routing table.

For each routing table entry, the command displays the route prefix, an on-link interface or a next-hop neighbor on an interface, a preference value (smaller is preferred), and a lifetime in seconds.

Routing table entries may also have **publish** and **aging** attributes. By default, they age (the lifetime counts down, rather than remaining constant) and are not published (not used in constructing router advertisements).

On hosts, routing table entries are normally auto-configured from router advertisements.

```
ipv6 rtu prefix if#[/nexthop] [lifetime L] [preference P] [publish] [age] [spl site-prefix-length]
```

Adds or removes a route in the routing table. The route prefix is required. The prefix can be on-link to a specified interface, or next-hop, specified with a neighbor address on an interface. The route can have a lifetime in seconds, with the default infinite, and a preference, with the default of zero, or most preferred. Specifying a lifetime of zero deletes the route.

If the route is specified as published, indicating it will be used in constructing router advertisements, it does not age. The route's lifetime does not count down and therefore is effectively infinite, but the value is used in router advertisements. Optionally, a route can be specified as a published route that also ages. A nonpublished route by default always ages.

The optional **spl** suboption can be used to specify a site prefix length associated with the route. The site prefix length is used only when sending router advertisements.

Lifetime can be abbreviated as **life**, preference as **pref**, and publish as **pub**.

# Ipsec6.exe

3/5/2021 • 2 minutes to read • [Edit Online](#)

Configuration of IPSec policies and security associations for IPv6 is done with Ipsec6.exe. The following are Ipsec6.exe subcommands:

**ipsec6 sp [*Interface*]**

Displays active security policies. Alternately, displays active security policies for a specific interface.

**ipsec6 sa**

Displays active security associations.

**ipsec6 c [*FileName*(*without extension*)]**

Creates files used to configure security policy and security associations. Creates *FileName*.spd for security policies and *FileName*.sad for security associations. Use the created files as templates to configure security policies or security associations. The files can be modified with a text editor. To invoke the configuration contained in the SPD and SAD files, use the **ipsec6 a** command.

**ipsec6 a [*FileName*(*without extension*)]**

Adds security policies in *FileName*.spd and security associations in *FileName*.sad to the list of active security policies and security associations.

**ipsec6 i [*Policy*] [*FileName*(*without extension*)]**

Adds security policies in *FileName*.spd and security associations in *FileName*.sad to the list of active security policies and security associations as referenced by policy number.

**ipsec6 d [type = sp sa] [*IndexNumber*]**

Deletes security policies or security associations from the list of active security policies and security associations, as referenced by their index number (displayed with **ipsec6 sp** or **ipsec6 sa**).

# Using Internet Explorer to Access IPv6 Websites

3/5/2021 • 2 minutes to read • [Edit Online](#)

Microsoft Internet Explorer supports using IPv6 to connect and access IPv6-enabled sites (web servers and FTP servers, for example) when the following circumstances are met:

- The host machine running Internet Explorer must have IPv6 installed and enabled.
  - When connecting to a host that is outside of the local subnet, the interface that provides the connectivity must have a routable IPv6 address assigned.
  - When connecting to the IPv6 loopback address (::1) or to a link-local destination, a routable IPv6 address is not required.
- If the requested URL contains a name ([www.contoso.com](http://www.contoso.com), for example), the Domain Name System (DNS) query for that name must return one or more IPv6 addresses.
- If the requested URL contains an IPv6 address (::1, for example), the IPv6 address must be surrounded by square brackets ([https://\[::1\]](http://[::1])). Scope IDs, sometimes called zone indexes, are supported as part of the IPv6 address. The scope ID is used to determine which network interface to use when sending the request on computers with multiple network interfaces. The scope ID is specified using the '%' character after the main IPv6 address (fe80::1%11, for example). However, the '%' character must be escaped in the URL used with Internet Explorer. For example, scope ID 11 on a link-local address would be specified as [https://\[fe80::1%2511\]](http://[fe80::1%2511]) when using Internet Explorer. This ability to use IPv6 addresses in a URL is supported on Internet Explorer 7 and later.

## NOTE

If Internet Explorer is configured to use a proxy server, the proxy server must have an IPv6 address assigned and the proxy server must be able to proxy IPv6 addresses. The proxy server would be used to connect to an external host using IPv6. The proxy server would normally be bypassed for the IPv6 loopback address and IPv6 link-local addresses.

# Recommended Configurations for IPv6

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following are recommended configurations to test network applications for IPv6:

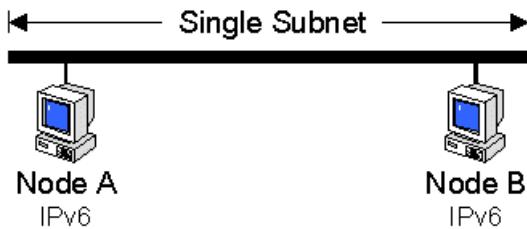
- [Single subnet with link-local addresses](#)
- [IPv6 traffic between nodes on different subnets of an IPv4 internetwork \(6to4\)](#)
- [Using IPsec between two local link hosts](#)

# Configuration 1: Single Subnet with Link-local Addresses

3/5/2021 • 3 minutes to read • [Edit Online](#)

The first configuration requires no additional configuration beyond installing the Microsoft IPv6 Technology Preview protocol. This configuration consists of at least two nodes on the same subnet. In IPv6 terminology, the two nodes are on the same link with no intermediate routers.

The following illustration shows the configuration of two nodes on a single subnet using link-local addresses.



By default, IPv6 configures link-local IP addresses for each interface corresponding to installed Ethernet network adapters. Link-local addresses have the prefix fe80::/64. The last 64 bits of the IPv6 address is known as the interface identifier and is derived from the 48-bit MAC address of the network adapter.

To create the IPv6 interface identifier from the 48-bit (6-byte) Ethernet MAC address:

- The hex digits 0xff-fe are inserted between the third and fourth byte of the MAC address.
- The Universal/Local bit, the second low-order bit of the first byte of the MAC address, is complemented. If it is a 1, it is turned to 0, and if it is a 0, it is turned to 1.

For example, for the MAC address 00-60-08-52-f9-d8:

- The hex digits 0xff-fe are inserted between 0x08 (the third byte) and 0x52 (the fourth byte) of the MAC address, forming the 64-bit address 00-60-08-ff-fe-52-f9-d8.
- The Universal/Local bit, the second low-order bit of 0x00 (the first byte) of the MAC address is complemented. The second low-order bit of 0x00 is 0 which, when complemented, becomes 1. The result is that for the first byte, 0x00 becomes 0x02.

Therefore, the IPv6 interface identifier corresponding to the Ethernet MAC address of 00-60-08-52-f9-d8 is 02-60-08-ff-fe-52-f9-d8.

The link-local address of a node is the combination of the prefix fe80::/64 and the 64-bit interface identifier expressed in IPv6 colon-hexadecimal notation. Therefore, the link-local address of this example node with the prefix fe80::/64 and the interface identifier 02-60-08-ff-fe-52-f9-d8 is fe80::260:8ff:fe52:f9d8.

You can view your link local address by using `ip6 if`, as demonstrated in the following example:

```
ip6 if
```

```

Interface 4 (site 1): Local Area Connection
  uses Neighbor Discovery
  link-level address: 00-10-5a-aa-20-a2
    preferred address fe80::210:5aff:feaa:20a2, infinite/infinite
    multicast address ff02::1, 1 refs, not reportable
    multicast address ff02::1:ffaa:20a2, 1 refs, last reporter
  link MTU 1500 (true link MTU 1500)
  current hop limit 128
  reachable time 43500ms (base 30000ms)
  retransmission interval 1000ms
  DAD transmits 1
Interface 3 (site 1): 6-over-4 Virtual Interface
  uses Neighbor Discovery
  link-level address: 10.0.0.2
    preferred address fe80::a00:2, infinite/infinite
    multicast address ff02::1, 1 refs, not reportable
    multicast address ff02::1:ff00:2, 1 refs, last reporter
  link MTU 1280 (true link MTU 65515)
  current hop limit 128
  reachable time 34000ms (base 30000ms)
  retransmission interval 1000ms
  DAD transmits 1
Interface 2 (site 0): Tunnel Pseudo-Interface
  does not use Neighbor Discovery
  link-level address: 0.0.0.0
    preferred address ::10.0.0.2, infinite/infinite
  link MTU 1280 (true link MTU 65515)
  current hop limit 128
  reachable time 0ms (base 0ms)
  retransmission interval 0ms
  DAD transmits 0
Interface 1 (site 0): Loopback Pseudo-Interface
  does not use Neighbor Discovery
  link-level address:
    preferred address ::1, infinite/infinite
  link MTU 1500 (true link MTU 1500)
  current hop limit 1
  reachable time 0ms (base 0ms)
  retransmission interval 0ms
  DAD transmits 0

```

Interface 4 is an interface corresponding to an installed Ethernet adapter with a link-local address of fe80::210:5aff:feaa:20a2.

For more information on IPv6 addressing and an overview of IPv6 concepts, see the [Introduction to IPv6 white paper](#).

## Testing Connectivity Between Two Link-local Hosts

You can do a simple ping (an exchange of ICMPv6 Echo Request and Echo Reply messages) using IPv6 between two link-local hosts.

### To ping using IPv6 between two link-local hosts

1. Install the Microsoft IPv6 Technology Preview for Windows on two Windows hosts (Host A and Host B) that are on the same link (subnet).
2. Use `ipv6` if on Host A to obtain the link-local address for the Ethernet interface.

Example: The link-local address of Host A is fe80::210:5aff:feaa:20a2.

3. Use `ipv6` if on Host B to obtain the link-local address for the Ethernet interface.

Example: The link-local address of Host B is fe80::260:97ff:fe02:6ea5.

4. From Host A, ping Host B using Ping6.exe.

Example: `ping6 fe80::260:97ff:fe02:6ea5`

To specify the source address from which the Echo Request messages are sent, you can also use the Ping6.exe -s option. For example, to send Echo Requests to Host B from the IPv6 address of fe80::210:5aff:fea:20a2, use the following command:

```
ping6 -s fe80::210:5aff:fea:20a2%4 fe80::260:97ff:fe02:6ea5
```

When pinging a link-local or site-local address, it is recommended to specify the scope-ID to make the destination address unambiguous. The notation to specify the scope-ID is address%scope-ID. For link-local addresses, the scope-ID is equal to the interface number as displayed in the `ip6 -f` command. For site-local addresses, the scope-ID is equal to the site number as displayed in the `ip6 -f` command. For example, to send Echo Request messages to Host B using scope-ID 4, use the following command:

```
ping6 fe80::260:97ff:fe02:6ea5%4
```

## Related topics

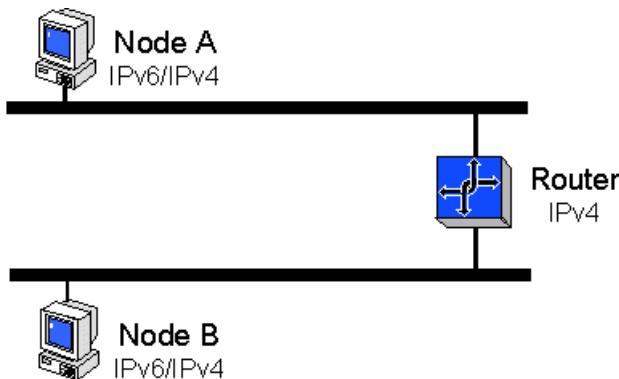
[Recommended Configurations for IPv6](#)

# Configuration 2: IPv6 Traffic Between Nodes on Different Subnets of an IPv4 Internetwork (6to4)

3/5/2021 • 7 minutes to read • [Edit Online](#)

6to4 is a method for connecting IPv6 hosts or sites over the existing IPv4 Internet infrastructure. It uses a unique address prefix to give isolated IPv6 sites their own IPv6 address space. 6to4 is like a "pseudo-ISP" providing IPv6 connectivity. You can use 6to4 to communicate directly with other 6to4 sites. 6to4 does not require the use of IPv6 routers and its IPv6 traffic is encapsulated with an IPv4 header.

The following illustration shows the configuration of two nodes on separate subnets using 6to4 to communicate across an IPv4 router.



The main requirement for using 6to4 is one globally routable IPv4 address for your site. Suppose that your site consists of a collection of IPv6 computers that you manage (some running the Microsoft IPv6 protocol and some running other IPv6 implementations). Assume also that all IPv6 computers are directly connected using Ethernet or 6-over-4. The globally routable IPv4 address must be assigned to one of your computers running the Microsoft IPv6 protocol. This computer will be your 6to4 gateway.

If you have an IPv4 address that is part of the private address space (10.0.0.0/8, 172.16.0.0/12, or 192.168.0.0/16) or the Automatic Private IP Addressing (APIPA) address space of 169.254.0.0/16 used by Windows 2000, it is not globally routable. Otherwise, it is probably a public IP address and is globally routable. See the [Debugging 6to4 Configuration](#) topic in this document for more help in determining whether your ISP connection supports 6to4.

## The 6to4cfg.exe Tool

The 6to4cfg.exe tool automates 6to4 configuration, automatically discovering your globally routable IPv4 address and creating a 6to4 prefix. It either performs the configuration directly, or it can write a configuration script that you can inspect and run later.

The basic 6to4cfg.exe command syntax is as follows.

```
6to4cfg [-r] [-s] [-u] [-R relay] [-b] [-S address] [filename]
```

[filename]

Writes the configuration script to a file, if you specify a file name. The configuration script uses Ipv6.exe.

You can specify con for the file name to write the configuration script to console output, which is useful for

seeing what 6to4cfg.exe will do in a test scenario.

If you do not specify a file name, 6to4cfg.exe directly updates the IPv6 configuration on your computer.

-r

Becomes a 6to4 gateway router for your local network, enabling routing on all of your interfaces and assigned subnet prefixes.

-s

Enables site-local addressing in your 6to4 site. This command is only useful when used in conjunction with -r.

-u

Specifies that the 6to4 configuration be reversed. Therefore 6to4cfg -u reverses the effect of 6to4cfg and 6to4cfg -r -u reverses the effect of 6to4cfg -r, and so on.

-R *relay*

Specifies the name or IPv4 address of a 6to4 relay router. The default name is 6to4.ipv6.microsoft.com, the 6to4 relay router on the Internet.

-b

Specifies that 6to4cfg.exe picks the "best" relay address instead of the first.

-S *address*

Specifies the local IPv4 address for the 6to4 prefix.

## Manual 6to4 Configuration

In this example the address of the 6to4 gateway is 172.31.42.239. You need your own globally routable IPv4 address to use 6to4.

### NOTE

The IP address 172.31.42.239 is used for example purposes only. This is a private address and is not globally routable on the Internet.

The 32-bit globally routable IPv4 address is combined with the 16-bit prefix 2002::/16 to form a 48-bit IPv6 address prefix for your site. In this example, the 6to4 site prefix is 2002:ac1f:2aef:/48. Note that ac1f:2aef is the hexadecimal encoding of 172.31.42.239 (of course, you will use a different prefix based on your own globally routable IPv4 address). Using the 6to4 site prefix, you can assign addresses and subnet prefixes inside your site.

This example assumes that you use subnet 0 for manually configuring a 6to4 address on your 6to4 gateway computer and that you use subnet 1 for automatically configuring addresses on your Ethernet network segment. However, other choices are possible.

1. Use Ipv6.exe to enable 6to4 on your 6to4 gateway computer:

```
ipv6 rtu 2002::/16 2
```

The **ipv6 rtu** command performs a routing table update. It can be used to add, remove, or update a route. In this case it is enabling 6to4.

The 2002::/16 argument is the prefix of the route, specifying the unique 6to4 prefix.

The 2 argument specifies the on-link interface for this prefix. Interface #2 is the "pseudo-interface" used for configured tunnels, automatic tunneling, and 6to4. When an IPv6 destination address matches the 2002::/16 prefix, the 32 bits that follow the prefix in the destination address are extracted to form an IPv4 destination address. The packet is encapsulated with an IPv4 header and sent to the IPv4 destination address.

2. Configure a 6to4 address on your 6to4 gateway computer:

```
ipv6 adu 2/2002:ac1f:2aef::ac1f:2aef
```

The **ipv6 adu** command performs an address update. It can be used to add, remove, or update an address on an interface. In this instance, it is configuring the computer's 6to4 address.

The 2/2002:ac1f:2aef::ac1f:2aef argument specifies both the interface and the address. It requires configuring address 2002:ac1f:2aef::ac1f:2aef on interface #2. The address is created using the site prefix 2002:ac1f:2aef::/48, subnet 0 to give a subnet prefix 2002:ac1f:2aef::/64, and a 64-bit interface identifier. The convention demonstrated uses the computer's IPv4 address for the interface identifier for addresses assigned to interface #2. For your use, ac1f:2aef should be replaced by the hexadecimal encoding of your own globally routable IPv4 address.

The two preceding commands are sufficient to allow communication with other 6to4 sites. For example, you can try pinging the Microsoft 6to4 site:

```
ping6 2002:836b:9820::836b:9820
```

To enable communication with the 6bone, you must create a default configured tunnel to a 6to4 relay. You can use the Microsoft 6to4 relay router, 131.107.152.32:

```
ipv6 rtu ::/0 2::131.107.152.32 pub life 1800
```

The **ipv6 rtu** command performs a routing table update, establishing, in this instance, a default route to the 6to4 relay.

The ::/0 argument is the route prefix. The zero-length prefix indicates that it is a default route.

The 2::131.107.152.32 argument specifies the next-hop neighbor for this prefix. It requires that packets matching the prefix are forwarded to address ::131.107.152.32 using interface #2. Forwarding a packet to ::131.107.152.32 on interface #2 causes it to be encapsulated with a v4 header and sent to 131.107.152.32.

The pub argument makes this a published route. Because this is only relevant for routers, it has no effect until routing is enabled. Similarly, the 30-minute lifetime pertains only if routing is enabled.

You should be able to access 6bone sites as well as 6to4 sites. You can use the following command to test this:

```
ping6 3ffe:1cff:0:f5::1
```

The final step is to enable routing on your 6to4 gateway. This example assumes that interface #3 on your gateway computer is an Ethernet interface and interface #4 is 6-over-4. Your computer might number its interfaces differently. The following two commands assign subnet prefixes to the two links. The subnet prefixes are derived from the site's 6to4 prefix 2002:ac1f:2aef::/48:

```
ipv6 rtu 2002:ac1f:2aef:1::/64 3 pub life 1800
```

```
ipv6 rtu 2002:ac1f:2aef:2::/64 4 pub life 1800
```

The **ipv6 rtu** command specifies that the prefix 2002:ac1f:2aef:1::/64 is on-link to interface #3. It is configuring the first subnet prefix on the Ethernet interface. The route is published with a lifetime of 30 minutes.

Similarly, the 2002:ac1f:2aef:2::/64 prefix is configured on the 6-over-4 interface.

The next three commands enable the 6to4 gateway computer to function as a router:

```
ipv6 ifc 2 forw
```

```
ipv6 ifc 3 forw adv
```

```
ipv6 ifc 4 forw adv
```

The **ipv6 ifc** command controls attributes of an interface. A router forwards packets and sends router advertisements. In the Microsoft IPv6 implementation, these per-interface attributes are controlled separately.

Interface #2 is not needed for advertising because it is a pseudo-interface.

If your computer has additional interfaces, they should also be configured to be forwarding and advertising.

After running these commands, the Microsoft IPv6 protocol will automatically configure addresses on interfaces #3 and #4 using the respective subnet prefixes and the two interfaces will start sending router advertisements at approximately 3 to 10 minute intervals.

Hosts receiving these router advertisements will automatically configure themselves with a default route and a 6to4 address derived from the subnet prefix of their link. They will have communication to other 6to4 sites and the 6bone through the gateway computer.

## Debugging 6to4 Configuration

### To debug your 6to4 configuration

1. Check your IPv4 connectivity to the 6to4 relay router:

```
ping 6to4.ipv6.microsoft.com
```

If this fails, then you do not have global Internet connectivity.

2. Check IPv6 encapsulation by using automatic tunneling:

```
ping6 ::131.107.152.32
```

If this fails, then you might have a firewall or network address translator (NAT) between your computer and the Internet. If this is successful, then your Internet connection can support 6to4.

3. Check the display of the **ipv6 rt** command. You should see a 2002::/16 -> 2 route. Check the display of the **ipv6 if 2** command. You should see a preferred address with a 2002::/16 prefix.

#### NOTE

The IPv4 address of the Microsoft 6to4 relay router of 131.107.152.32 is subject to change. If Step 2 above does not work, check the output of the ping command in step 1 to verify the IPv4 address of the Microsoft 6to4 relay router.

## Related topics

[Recommended Configurations for IPv6](#)

[Single subnet with link-local addresses](#)

[Using IPsec between two local link hosts](#)

# Configuration 3: Using IPsec Between Two Local-link Hosts

3/5/2021 • 5 minutes to read • [Edit Online](#)

This configuration creates an IPsec Security Association (SA) between two hosts on the same subnet to perform authentication using the Authentication Header (AH) and the Message Digest 5 (MD5) hashing algorithm. In this example, the configuration shown secures all traffic between two neighboring hosts: Host 1, with the link-local address FE80::2AA:FF:FE53:A92C, and Host 2, with the link-local address FE80::2AA:FF:FE92:D0F1.

## To use IPsec between two local-link hosts

1. On Host 1, create blank security association (SAD) and security policy (SPD) files by using the ipsec6 c command. In this example, the Ipsec6.exe command is ipsec6 c test. This creates two files to manually configure security associations (Test.sad) and security policies (Test.spd).
2. On Host 1, edit the SPD file to add a security policy that secures all traffic between Host 1 and Host 2.

The following table shows the security policy added to the Test.spd file before the first entry for this example (the first entry in the Test.spd file was not modified).

SPD FILE FIELD NAME	EXAMPLE VALUE
Policy	2
RemoteIPAddr	FE80::2AA:FF:FE92:D0F1
LocalIPAddr	*
RemotePort	*
Protocol	*
LocalPort	*
IPSecProtocol	AH
IPSecMode	TRANSPORT
RemoteGWIPAddr	*
SABundleIndex	NONE
Direction	BIDIRECT
Action	APPLY
InterfaceIndex	0

Place a semicolon at the end of the line configuring this security policy. The policy entries must be placed

in decreasing numerical order.

3. On Host 1, edit the SAD file, adding SA entries to secure all traffic between Host 1 and Host 2. Two security associations must be created, one for traffic to Host 2 and one for traffic from Host 2.

The following table shows the first SA entry added to the Test.sad file for this example (for traffic to Host 2).

SAD FILE FIELD NAME	EXAMPLE VALUE
SAEntry	2
SPI	3001
SDADestIPAddr	FE80::2AA:FF:FE92:D0F1
DestIPAddr	POLICY
SrcIPAddr	POLICY
Protocol	POLICY
DestPort	POLICY
SrcPort	POLICY
AuthAlg	HMAC-MD5
KeyFile	Test.key
Direction	OUTBOUND
SecPolicyIndex	2

Place a semicolon at the end of the line configuring this SA.

The following table shows the second SA entry added to the Test.sad file for this example (for traffic from Host 2).

SAD FILE FIELD NAME	EXAMPLE VALUE
SAEntry	1
SPI	3000
SDADestIPAddr	FE80::2AA:FF:FE53:A92C
DestIPAddr	POLICY
SrcIPAddr	POLICY
Protocol	POLICY

SAD FILE FIELD NAME	EXAMPLE VALUE
DestPort	POLICY
SrcPort	POLICY
AuthAlg	HMAC-MD5
KeyFile	Test.key
Direction	INBOUND
SecPolicyIndex	2

Place a semicolon at the end of the line configuring this SA. The SA entries must be placed in decreasing numerical order.

- On Host 1, create a text file that contains a text string used to authenticate the SAs created with Host 2. In this example, the file Test.key is created with the contents "This is a test". You must include double quotes around the key string in order for the key to be read by the ipsec6 tool.

The Microsoft IPv6 Technology Preview only supports manually configured keys for the authentication of IPsec SAs. The manual keys are configured by creating text files that contain the text string of the manual key. In this example, the same key for the SAs is used in both directions. You can use different keys for inbound and outbound SAs by creating different key files and referencing them with the KeyFile field in the SAD file.

- On Host 2, create blank security association (SAD) and security policy (SPD) files by using the ipsec6 c command. In this example, the Ipsec6.exe command is ipsec6 c test. This creates two files with blank entries for manually configuring security associations (Test.sad) and security policies (Test.spd).

To simplify the example, the same file names for the SAD and SPD files are used on Host 2. You can choose to use different file names on each host.

- On Host 2, edit the SPD file to add a security policy that secures all traffic between Host 2 and Host 1.

The following table shows the security policy entry added before the first entry to the Test.spd file for this example (the first entry in the Test.spd file was not modified).

SPD FILE FIELD NAME	EXAMPLE VALUE
Policy	2
RemoteIPAddr	FE80::2AA:FF:FE53:A92C
LocalIPAddr	*
RemotePort	*
Protocol	*
LocalPort	*

SPD FILE FIELD NAME	EXAMPLE VALUE
IPSecProtocol	AH
IPSecMode	TRANSPORT
RemoteGWIPAddr	*
SABundleIndex	NONE
Direction	BIDIRECT
Action	APPLY
InterfaceIndex	0

Place a semicolon at the end of the line configuring this security policy. The policy entries must be placed in decreasing numerical order.

- On Host 2, edit the SAD file, adding SA entries to secure all traffic between Host 2 and Host 1. Two security associations must be created—one for traffic to Host 1 and one for traffic from Host 1.

The following table shows the first SA added to the Test.sad file for this example (for traffic from Host 1).

SAD FILE FIELD NAME	EXAMPLE VALUE
SAEntry	2
SPI	3001
SADestIPAddr	FE80::2AA:FF:FE92:D0F1
DestIPAddr	POLICY
SrcIPAddr	POLICY
Protocol	POLICY
DestPort	POLICY
SrcPort	POLICY
AuthAlg	HMAC-MD5
KeyFile	Test.key
Direction	INBOUND
SecPolicyIndex	2

Place a semicolon at the end of the line configuring this SA.

The following table shows the second SA entry added to the Test.sad file for this example (for traffic to Host 1).

SAD FILE FIELD NAME	EXAMPLE VALUE
SAEntry	1
SPI	3000
SADestIPAddr	FE80::2AA:FF:FE53:A92C
DestIPAddr	POLICY
SrcIPAddr	POLICY
Protocol	POLICY
DestPort	POLICY
SrcPort	POLICY
AuthAlg	HMAC-MD5
KeyFile	Test.key
Direction	OUTBOUND
SecPolicyIndex	2

Place a semicolon at the end of the line configuring this SA. The SA entries must be placed in decreasing numerical order.

8. On Host 2, create a text file that contains a text string used to authenticate the SAs created with Host 1. In this example, the file Test.key is created with the contents "This is a test". You must include double quotes around the key string in order for the key to be read by the ipsec6 tool.
9. On Host 1, add the configured security policies and SAs from the SPD and SAD files using the ipsec6 a command. In this example, the ipsec6 a test command is run on Host 1.
10. On Host 2, add the configured security policies and SAs from the SPD and SAD files by using the ipsec6 a command. In this example, the ipsec6 a test command is run on Host 2.
11. Ping Host 1 from Host 2 with the ping6 command.

If you capture the traffic using Microsoft Network Monitor or another packet sniffer, you should see the exchange of ICMPv6 Echo Request and Echo Reply messages with an Authentication Header between the IPv6 header and the ICMPv6 header.

## Related topics

[Recommended Configurations for IPv6](#)

Single subnet with link-local addresses

IPv6 traffic between nodes on different subnets of an IPv4 internetwork (6to4)

# Additional IPv6 Topics

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following topics provide additional information related to the Microsoft IPv6 Technology Preview protocol.

- [IPv6 Router Advertisements](#)
- [IPv6 Site Prefixes](#)
- [IPv6 Link-local and Site-local Addresses](#)
- [IPv6 Multicast Destination Addresses](#)
- [IPv6 Automatic Tunneling and 6to4](#)
- [IPv6 Forwarding Tunneled Packets](#)

# IPv6 Router Advertisements

3/5/2021 • 2 minutes to read • [Edit Online](#)

The content of IPv6 router advertisements is automatically derived from the published routes in the routing table. Nonpublished routes are used for routing but are ignored when constructing router advertisements.

Router advertisements for IPv6 always contain a source link-layer address option and an MTU option. The value for the MTU option is taken from the sending interface's current link MTU. This value can be changed with the `ipv6 ifc mtu` command.

The router advertisement only has a nonzero router lifetime if there is a published default route. A default route is a route for the zero-length prefix.

Published on-link routes result in prefix information options in router advertisements. If the on-link prefix has 64 bits, the prefix information option has both the L and A bits set and hosts receiving it will autoconfigure addresses.

An interface that sends router advertisements will also autoconfigure addresses for itself based on the prefix information options that it sends.

A finite nonaging lifetime on all published routes (for example, 30 minutes) is recommended. If you decide to retract a route, you can change the route to have an aging lifetime. The route will age over the course of several router advertisements and then disappear from both the router and any hosts receiving the router advertisements.

Routes that hosts find through router advertisements age and are not published. Addresses autoconfigured from router advertisements age as well.

# IPv6 Site Prefixes

3/5/2021 • 2 minutes to read • [Edit Online](#)

The `ipv6 rtu` command allows published on-link prefixes to be configured with a site prefix length. If specified, the site prefix length is put into a prefix information option in router advertisements. For example:

```
ipv6 rtu 2002:836b:9820:2::/64 4 pub life 1800 spl 48
```

specifies a prefix that is on-link to interface #4. The prefix is published, meaning that it is included in router advertisements if interface #4 is an advertising interface. The lifetime in the prefix information option is 1800 seconds (30 minutes). The site prefix length in the prefix information option is 48.

The receipt of a prefix information option that specifies a site prefix causes an entry to be created in the site prefix table, which can be displayed by using the `ipv6 spt` command. The site prefix table is used to remove inappropriate site-local addresses from those returned by the [getaddrinfo](#) and related functions.

## Related topics

[IPv6 Link-local and Site-local Addresses](#)

[Netsh.exe](#)

# IPv6 Link-local and Site-local Addresses

3/5/2021 • 4 minutes to read • [Edit Online](#)

IPv6 link-local and site-local addresses are called scoped addresses. The Windows Sockets (Winsock) API supports the `sin6_scope_id` member in the `sockaddr_in6` structure for use with scoped addresses. For IPv6 link-local addresses (fe80::/10 prefix), the `sin6_scope_id` member in the `sockaddr_in6` structure is the interface number. For IPv6 site-local addresses (fec0::/10 prefix), the `sin6_scope_id` member in the `sockaddr_in6` structure is a site identifier.

An example of a link-local IPv6 address on interface #5 is the following:

```
fe80::208:74ff:fed:a625c%5
```

The following command is available on Windows XP with Service Pack 1 (SP1) and later to query and configure IPv6 on a local computer:

- [Netsh.exe](#)

Configuration changes made using the Netsh.exe commands are permanent and are not lost when the computer or the IPv6 protocol is restarted.

Prior to Windows XP with Service Pack 1 (SP1), IPv6 configuration and management used several older command-line tools (Net.exe, Ipv6.exe, and Ipsec6.exe) to configure and manage IPv6. Using these older tools, the IPv6 changes are not permanent and are lost when the computer or the IPv6 protocol was restarted. These older command-line tools are only supported on Windows XP.

On Windows XP with SP1, the following command will display the list of IPv6 interfaces on a local computer including the interface index, the interface name, and various other interface properties.

```
netsh interface ipv6 show interface
```

On Windows XP with SP1, the following command will change the site identifier associated with an interface index.

```
netsh interface ipv6 set interface siteid=value
```

On Windows XP, the following older command will also change the site identifier associated with a site-local address to 3.

```
ipv6 rtu fec0::/10 3
```

If you are sending or connecting to a scoped address, then the `sin6_scope_id` member in the `sockaddr_in6` structure can be left unspecified (zero) which represents an ambiguous scoped address. For example, the following link-local address is ambiguous:

```
fe80::10
```

If you are binding to a scoped address, then the `sin6_scope_id` member in the `sockaddr_in6` structure must contain a nonzero value that specifies a valid interface number for a link-local address or a site identifier for a site-local address.

## Ambiguous Scoped Addresses

If you are sending or connecting to a scoped address and have not specified the `sin6_scope_id` member in the `sockaddr_in6` structure, then the scoped address is ambiguous. To resolve this, the IPv6 protocol first determines whether you have bound the socket to a source address. If so, the bound source address resolves the ambiguity by supplying an interface number or site identifier.

If you are sending or connecting to a scoped address and have neither specified the `sin6_scope_id` member nor bound a source address, then the IPv6 protocol checks the routing table. For example, the following command will display the IPv6 routing table on a local computer:

```
netsh interface ipv6 show route
```

No	Manual	256	fe80::/64	13	Local Area Connection
No	Manual	256	fe80::/64	14	Wireless Network Connection

This indicates that link-local addresses are treated as on-link to interface #13 and #14 by default.

Ambiguity arises when a local computer has multiple network adapters. For example, the `netsh` command above indicates there are two network interfaces (Local Area Connection and Wireless Network Connection). When an application specifies a destination link-local address (fe80::10, for example) without a scope ID, it is not clear which adapter to use to send the packet. Only a link-local unicast (fe80::/64) or a link-scope multicast (ff00::/8) IPv6 destination address can suffer from not having a scope ID when sending a packet.

## Neighbor Discovery

If you have not specified the `sin6_scope_id` member in the `sockaddr_in6` structure, have not bound a source address, and have not specified a route for link-local addresses, then the IPv6 protocol will try Neighbor Discovery to resolve the destination link-local address. For a given packet being sent, one interface is tried. This first interface that is tried is considered the most preferred interface. If Neighbor Discovery fails to resolve the link-local address on an interface, the packet to be sent is dropped and the system remembers that the destination link-local address is not reachable over that interface. On the next packet to be sent under all of the same conditions, a different interface is tried for Neighbor Discovery. This process continues through each of the interfaces on a local computer for each new packet until Neighbor Discovery responds for the destination link-local address or all of the possible interfaces have been tried and failed. Each time an attempt to resolve the neighbor fails, one interface is eliminated for that neighbor.

If the destination link-local address resolves, then that interface is used to send the current packet. This interface is also used for any subsequent ambiguously-scoped packets that are sent to the same link-local destination address.

If Neighbor Discovery fails to resolve the destination link-local address on all interfaces, the system then tries to send the packet on the most preferred interface (the first interface tried). The network stack keeps trying to resolve the destination link-local address on the most preferred interface. After a period of time after Neighbor Discovery has failed on all interfaces, the network stack will restart the process again and try to resolve the destination link-local address on all of the interfaces. Currently, this time interval when Neighbor Discovery is again tried on all interfaces is 60 seconds. However, this time interval may change on versions of Windows and should not be assumed by an application.

### NOTE

If an application binds the same link-local address to a different interface after Neighbor Discovery has resolved the link-local address, that will not override the interface with the link-local destination address returned by Neighbor Discovery.

For more information on Neighbor Discovery for IP version 6, see [RFC4861](#) published by the IETF.

## Related topics

[IPv6 Site Prefixes](#)

[Ipv6.exe](#)

[Netsh.exe](#)

[Using IPv6](#)

# IPv6 Multicast Destination Addresses

3/5/2021 • 2 minutes to read • [Edit Online](#)

When sending to a multicast destination address, the Microsoft IPv6 protocol normally requires that the application have an outgoing interface specified. This is done with the **IPV6\_MULTICAST\_IF** socket option or by binding the socket to a specific source address.

You can also specify a default interface for a specific multicast address, an entire multicast address scope, or all multicast addresses. This is done with a route that places the multicast prefix on-link to the desired outgoing interface. For following examples show how this can be specified:

```
ipv6 rtu ff02::5/128 3
ipv6 rtu ff0e::/16 3
ipv6 rtu ff00::/8 3
```

# IPv6 Automatic Tunneling and 6to4

3/5/2021 • 2 minutes to read • [Edit Online](#)

Automatic tunneling with IPv4-compatible addresses and 6to4 both work through a route for a prefix that is on-link to interface #2. The 32 bits following the prefix are extracted and used as an IPv4 destination address for the encapsulated packet.

Automatic tunneling uses the ::/96 prefix, which is enabled by default. It can be disabled by removing the route for ::/96.

6to4 uses the 2002::/16 prefix, which is not enabled by default.

# IPv6 Forwarding Tunneled Packets

3/5/2021 • 2 minutes to read • [Edit Online](#)

There is a limitation in the code that receives encapsulated (tunneled) packets and demultiplexes them to a specific interface. If you want to forward tunneled packets received through a configured tunnel, then it is necessary to enable forwarding on any 6-over-4 interfaces as well as interface #2. Just enabling forwarding on interface #2 will not work. Typically when configuring a router, you would enable forwarding on all interfaces except loopback.

# Network Location Awareness Service Provider (NLA)

3/5/2021 • 2 minutes to read • [Edit Online](#)

Personal computers running Microsoft Windows often have numerous network connections, such as multiple network interface cards (NIC) connected to different networks, or a physical network connection and a dial-up connection. Windows Sockets has been capable of enumerating available network interfaces for some time, but certain critical information about network connections was previously unavailable. This includes information such as the logical network to which a Windows computer is attached or whether multiple interfaces are connected to the same network.

The Network Location Awareness service provider, commonly referred to as NLA, enables Windows Sockets 2 applications to identify the logical network to which a Windows computer is attached. In addition, NLA enables Windows Sockets applications to identify to which physical network interface a given application has saved specific information. NLA is implemented as a generic Windows Sockets 2 Name Resolution service provider.

# The Role of NLA

3/5/2021 • 2 minutes to read • [Edit Online](#)

The Network Location Awareness (NLA) service provider is vital for computers or devices that might move between different networks, and for selecting optimal configurations when more than one is available. For example, a wireless computer roaming between physical networks can use NLA to determine the proper configuration based on information about its available network connection. NLA also proves valuable when a multihomed computer has a physical connection to one network while also connected to another network through a dial-up connection or a tunnel.

In the past, developers had to obtain information about a logical network interface, and therefore make decisions about network connectivity, based on a multitude of disparate network information. In those circumstances, developers had to choose the appropriate network interface based on the IP address, the subnet of the interface, the Domain Name System (DNS) name associated with the interface, the MAC address of a NIC, a wireless network name, or other network information. NLA alleviates this problem by supplying a standard interface for enumerating logical network attachment information, correlating it with physical network interface information, and then providing notification when previously returned information gets invalidated.

NLA provides the following network location information:

## Logical Network Identity

NLA first attempts to identify a logical network by its DNS domain name. If a logical network does not have a domain name, NLA identifies the network from custom static information stored in the registry, and finally from its subnet address.

## Logical Network Interfaces

For each network to which a computer is attached, NLA supplies an *AdapterName* that uniquely identifies a physical interface such as a NIC, or a logical interface such as a RAS connection. The AdapterName can then be used with functions available in the IP Helper API to obtain further interface characteristics.

NLA implements the logical network as a service class, with an associated class GUID and properties. Each logical network for which NLA returns information is an instance of that service class.

# Querying NLA

3/5/2021 • 4 minutes to read • [Edit Online](#)

To obtain notification of invalidated logical networks, use the [WSANSPIoclt](#) function to register for network location change events. Two methods can be used to determine whether a previously valid network location has become invalid: polling methods, or notification using overlapped I/O or Windows messaging.

Queries are formed using the [WSALookupServiceBegin](#), [WSALookupServiceNext](#), and [WSALookupServiceEnd](#) functions to enumerate all available logical networks. The use of each of these functions is explained individually throughout the remainder of this section, beginning with the [WSALookupServiceBegin](#) function.

## NOTE

NLA requires the Mswsock.h header file, which by default is not included in the Winsock2.h file.

## Step 1: Initiate the Query

For quick reference, the [WSALookupServiceBegin](#) function has the following syntax:

```
INT WSALookupServiceBegin(
    LPWSAQUERYSET lpqsRestrictions,
    DWORD dwControlFlags,
    LPHANDLE lphLookup
);
```

NLA supports the following *dwControlFlags* lookup flags:

LUP\_RETURN\_NAME LUP\_RETURN\_COMMENT LUP\_RETURN\_BLOB LUP\_RETURN\_ALL LUP\_DEEP

These flags restrict the result sets returned in subsequent calls to the [WSALookupServiceNext](#) function to networks that contain fields of the specified type. For example, specifying LUP\_RETURN\_BLOB in the *dwControlFlags* parameter of the [WSALookupServiceBegin](#) function call restricts result sets from subsequent calls to [WSALookupServiceNext](#), to networks that contain BLOB information. Using the LUP\_RETURN\_ALL flag is equivalent to specifying LUP\_RETURN\_NAME, LUP\_RETURN\_COMMENT, and LUP\_RETURN\_BLOB, but not LUP\_DEEP.

For an explanation of these lookup flags, consult the [WSALookupServiceBegin](#) function reference page.

The lookup handle returned by NLA in the *lphLookup* parameter is private to NLA, and should not be modified. Since the returned handle is private to NLA, functions such as [WSAGetOverlappedResult](#) are not available.

NLA returns zero upon successful completion, as defined in the [WSALookupServiceBegin](#) function reference page. Otherwise, NLA supports the following error codes.

ERROR	MEANING
WSANOTINITIALISED	A successful call to the <a href="#">WSAStartup</a> function to initialize NLA was not performed.

ERROR	MEANING
WSAEINVAL	One or more parameters were invalid, or parameters specified in the function call apply to protocols other than IP.
WSASERVICE_NOT_FOUND	The <i>lpServiceClassId</i> parameter of the <a href="#">WSAQUERYSET</a> structure passed in the <i>lpqsRestrictions</i> parameter contains an invalid GUID.
WSANO_DATA	The LUP_CONTAINERS flag was specified in <i>dwControlFlags</i> parameter.
WSAEFAULT	An access violation occurred when attempting to access user-supplied parameters.
WSASYSNOTREADY	The NLA service is unavailable to process the request.
WSA_NOT_ENOUGH_MEMORY	NLA or the NLA service was unable to allocate enough memory to process this request.

## Step 2: Perform the Query

The next step in querying NLA requires use of the [WSALookupServiceNext](#) function. For quick reference, the [WSALookupServiceNext](#) function has the following syntax:

```
INT WSALookupServiceNext(
    HANDLE hLookup,
    DWORD dwControlFlags,
    LPDWORD lpdwBufferLength,
    LPWSAQUERYSET lpqsResults
);
```

The *hLookup* parameter is the lookup handle returned from the previous call to the [WSALookupServiceBegin](#) function.

The *dwControlFlags* parameter supports the following flags:

LUP\_RETURN\_NAME LUP\_RETURN\_COMMENT LUP\_RETURN\_BLOB LUP\_RETURN\_ALL  
LUP\_FLUSHPREVIOUS

These flags are independent of the flags supported in the [WSALookupServiceBegin](#) function call. Note that any constraints specified in the previous call to the [WSALookupServiceBegin](#) function constrain the lookup; adding flags with the [WSALookupServiceNext](#) function in an attempt to broaden the query beyond the constraints specified in the [WSALookupServiceBegin](#) call are silently ignored. Specifying a more restrictive set of flags than that specified in the [WSALookupServiceBegin](#) call, however, is allowed.

If the network detailed in *lpqsResults* is an active network, a series of **NLA\_BLOB** structures is appended as specified in the **lpBlob** member of the [WSAQUERYSET](#) structure returned in *lpqsResults*. These **NLA\_BLOB** structures may be chained, and can be enumerated by traversing the list while **NLA\_BLOB.header.nextOffset** is nonzero. To obtain results for all network location information, continue calling the [WSALookupServiceNext](#), function until the **WSA\_E\_NO\_MORE** error is returned, as explained in the [WSALookupServiceNext](#), reference page.

The [WSALookupServiceNext](#) function is also used in conjunction with the [WSANSPIctl](#) function to receive

notification of network changes. See [Notification from NLA](#) for more information.

NLA returns zero upon successful completion. Clients of NLA should continue calling the [WSALookupServiceNext](#) function until WSA\_E\_NO\_MORE is returned, indicating that all of the information about available networks has been returned.

Otherwise, calling the [WSALookupServiceNext](#) function for NLA supports the following error codes.

ERROR	MEANING
WSANOTINITIALISED	A successful call to the <a href="#">WSAStartup</a> function that initialized NLA was not performed.
WSA_INVALID_HANDLE	The lookup handle provided in the <i>hLookup</i> parameter was not a valid NLA SP handle. Clients must first call the <a href="#">WSALookupServiceBegin</a> function and receive a valid NLA SP handle to obtain query results.
WSAESYSNOTREADY	The NLA service is unavailable to process this request.
WSAEFAULT	The buffer size specified in the <i>lpdwBufferLength</i> parameter was insufficient to hold the results pointed to by <i>lpqsResults</i> . The required buffer is specified in <i>lpdwBufferLength</i> ; if the client cannot supply a sufficiently large buffer, the client can call the <a href="#">WSALookupServiceNext</a> function with <i>dwControlFlags</i> set to LUP_FLUSHPREVIOUS to skip the entry.
WSA_NOT_ENOUGH_MEMORY	NLA is unable to obtain network information from the NLA system service due to insufficient memory in the calling process.
WSA_E_NO_MORE	There are no additional networks to enumerate for the query.

## Step 3: Terminate the Query

When all queries to NLA are complete and an application no longer requires use of NLA, a call to the [WSALookupServiceEnd](#) function should be made. Do not call [WSALookupServiceEnd](#) if the application will receive change notification based on the submitted query. See [Notification from NLA](#) for more information on receiving notification. Like most Windows Sockets service providers, NLA maintains a reference count for its clients. Calling the [WSALookupServiceEnd](#) function when queries to NLA are completed enables system resources no longer required by NLA to be freed.

NLA supports the following error codes for [WSALookupServiceEnd](#) function calls.

ERROR	MEANING
WSANOTINITIALISED	A successful call to the <a href="#">WSAStartup</a> function to initialize NLA was not performed.
WSA_INVALID_HANDLE	The handle provided in the <i>hLookup</i> parameter was not a valid NLA SP handle.



# Notification from NLA

3/5/2021 • 2 minutes to read • [Edit Online](#)

NLA is capable of providing its clients with notification of network location changes. The mechanism used to request notification of change events is a combination of the [WSALookupServiceBegin](#), [WSANSPIoclt](#), and [WSALookupServiceNext](#) functions.

In order to receive change notification from NLA, a client must first call the [WSALookupServiceBegin](#) to obtain a valid NLA SP lookup handle. Next, the client can call [WSALookupServiceNext](#) or [WSANSPIoclt](#) in any order; to register for notification, call the [WSANSPIoclt](#) function with the SIO\_NSP\_NOTIFY\_CHANGE control code set in the *dwControlCode* parameter.

The [WSALookupServiceNext](#) function returns WSA\_E\_NO\_MORE as a set delimiter. When a client has registered for notification using the [WSANSPIoclt](#) function and [WSALookupServiceNext](#) returns WSA\_E\_NO\_MORE, calling [WSALookupServiceNext](#) again reveals whether a change has occurred:

- If no changes have occurred since the previous WSA\_E\_NO\_MORE, WSA\_E\_NO\_MORE is returned.
- If a change has occurred, or if a change has occurred and a polling call is made, the [WSALookupServiceNext](#) function call returns networks as [WSAQUERYSET](#) structures, with one of the following flags set in its *dwOutputFlags* member:

RESULT\_IS\_ADDED RESULT\_IS\_CHANGED RESULT\_IS\_DELETED

Change notification is provided for any fields that changed since the NLA lookup handle was obtained with the [WSALookupServiceBegin](#) function call, or since the last enumeration that resulted in the WSA\_E\_NO\_MORE error. When all changed network location information is provided, WSA\_E\_NO\_MORE is returned. Clients can reissue a [WSANSPIoclt](#) function call on the same query handle at any time, one at a time, with the SIO\_NSP\_NOTIFY\_CHANGE flag set. This capability enables a client to recycle the query handle on an ongoing basis, thereby relieving the client from having to maintain change-state information on its own. Once a client no longer needs change notifications, it should close the query handle using the [WSALookupServiceEnd](#) function.

# Registering a Service Instance with NLA

3/5/2021 • 2 minutes to read • [Edit Online](#)

NLA clients can record network configuration information on a system-wide basis, enabling future queries to return the specified configuration information regardless of whether the network is active. This capability allows NLA clients to affect a consistent network information user experience across multiple applications.

## Parameters

To register a service instance with the Network Location Awareness service provider, use the [WSASetService](#) function. In order to properly register a service instance certain parameters of the [WSASetService](#) function must be set with the appropriate information, as explained in this section. For quick reference, the [WSASetService](#) function has the following syntax:

```
INT WSASetService(
    LPWSAQUERYSET lpqsRegInfo,
    WSAESETSERVICEOP essOperation,
    DWORD dwControlFlags
);
```

For the *lpqsRegInfo* parameter, provide a [WSAQUERYSET](#) structure from either an NLA SP query result, or constructed adhering to the requirements of an NLA SP query, as specified in [Querying NLA](#).

Operations supported for the *essOperation* parameter are the following:

### RNRSERVICE\_REGISTER

The network defined in the [WSAQUERYSET](#) structure provided in *lpqsRegInfo* is made persistent for the active user by storing the network instance in the registry hive for the current user, which allows impersonation.

### RNRSERVICE\_DELETE

If the network defined in the [WSAQUERYSET](#) structure provided in *lpqsRegInfo* is persistent, it will be removed.

The operation specified in the *essOperation* parameter can be modified by the following options, which can be specified with binary OR logic:

### NLA\_FRIENDLY\_NAME

When used with RNRSERVICE\_REGISTER, the *lpszComment* field of the network defined in *lpqsRegInfo* is checked for validity and stored persistently. When used with RNRSERVICE\_DELETE and the defined network has a friendly name, the friendly name is removed but the network entry is left intact.

### NLA\_ALLUSERS\_NETWORK

When used with RNRSERVICE\_REGISTER, the entry is stored persistently under HKEY\_LOCAL\_MACHINE, making it available during queries to all users on the local computer. When used with RNRSERVICE\_DELETE, the specified network is removed from HKEY\_LOCAL\_MACHINE. An error is returned if the specified network is not present. In order to delete a network from the registry hive of the current user, this flag must not be specified. This flag is only valid in the security context of a local system administrator.

NLA supports the following error codes for [WSASetService](#) function calls:

ERROR	MEANING
WSANOTINITIALISED	A successful call to the <a href="#">WSAStartup</a> function to initialize NLA was not performed.
WSAEACCESS	NLA_ALLUSERS_NETWORK was specified in <i>dwControlFlags</i> while not in the security context of a local system administrator.
WSAEALREADY	The specified network is already persistently stored in the requested manner, and no flags were specified in <i>dwControlFlags</i> to indicate an update to the existing entry.
WSAEAFNOSUPPORT	A protocol family was specified for which there is no support. Only IP protocol families are supported in NLA.
WSAEPFNOSUPPORT	A protocol was specified for which there is no support. Only IP protocol is supported in NLA.

# About the Winsock SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

Winsock provides a Service Provider Interface for creating Winsock services, commonly referred to as the Winsock SPI. Two types of service providers exist: transport providers and namespace providers. Examples of transport providers include protocol stacks such as TCP/IP or IPX/SPX, while an example of a namespace provider would be an interface to the Internet's Domain Naming System (DNS). Separate sections of the service provider interface specification apply to each type of service provider.

[Transport](#) and [namespace](#) service providers must be registered with the Ws2\_32.dll at the time they are installed. This registration need only be done once for each provider as the necessary information is retained in persistent storage.

# Function Interface Model

3/5/2021 • 2 minutes to read • [Edit Online](#)

Windows Sockets transport and namespace-service providers are DLLs with a single exported procedure entry point for the service provider initialization function [WSPStartup](#) or [NSPStartup](#), respectively. All other service provider functions are made accessible to the Ws2\_32.dll through the service provider's dispatch table. Service provider DLL's are loaded into memory by the Ws2\_32.dll only when needed, and are unloaded when their services are no longer required.

The SPI also defines several circumstances in which a transport service provider calls up into the Ws2\_32.dll (upcalls) to obtain DLL support services. The transport service provider DLL is given the Ws2\_32.dll's upcall dispatch table through the *UpcallTable* parameter to [WSPStartup](#).

Service providers should have their file name extension changed from "DLL" to ".WSP" or ".NSP". This requirement is not strict. A service provider will still operate with the Ws2\_32.dll with any file name extension.

The Winsock SPI uses the following function prefix naming convention:

PREFIX	MEANING	DESCRIPTION
WSP	Windows Sockets Service Provider	Transport service provider entry points
WPU	Windows Sockets Provider Upcall	Ws2_32.dll entry points for service providers
WSC	Windows Sockets Configuration	WS2_32.dll entry points for installation applets
NSP	Namespace Provider	Namespace provider entry points

As described above, these entry points are not exported (with the exception of [WSPStartup](#) and [NSPStartup](#)), but are accessed through an exchange of dispatch tables.

# Service Provider Ordering

3/5/2021 • 2 minutes to read • [Edit Online](#)

The order in which transport service providers are initially installed governs the order in which they are enumerated through [WSCEnumProtocols](#) and [WSCEnumProtocols32](#) at the service provider interface, or through [WSAEnumProtocols](#) at the application interface. More importantly, this order also governs the order in which protocols and service providers are considered when a client requests creation of a socket based on its address family, type, and protocol identifier.

Windows Sockets 2 includes an applet called Sporder.exe that allows the catalog of installed protocols to be reordered interactively after protocols have already been installed. Winsock also includes an auxiliary DLL, Sporder.dll, that exports a procedural interface for reordering protocols. This procedural interface consists of a single procedure called [WSCWriteProviderOrder](#).

The interface definition may be imported into a C or C++ program by means of the include file Sporder.h. The entry point may be linked by means of the lib file Sporder.lib.

# Transport Service Providers

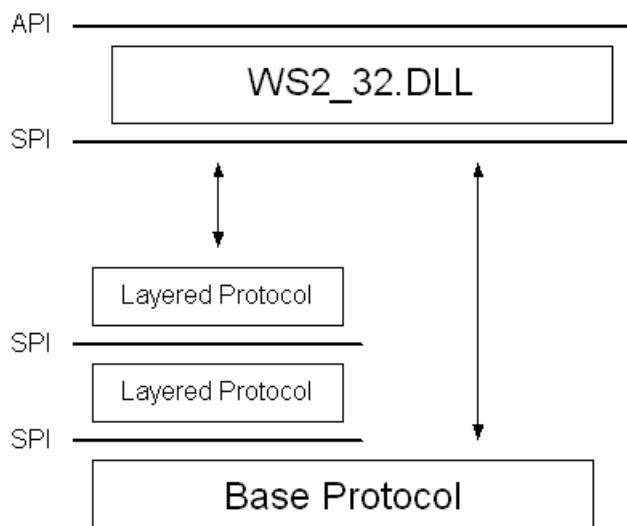
3/5/2021 • 2 minutes to read • [Edit Online](#)

A given transport service provider supports one or more protocols. For example, a TCP/IP provider would supply, as a minimum, the TCP and UDP protocols, while an IPX/SPX provider might supply IPX, SPX, and SPX II. Each protocol supported by a particular provider is described by a [WSAPROTOCOL\\_INFO](#) structure, and the total set of such structures can be thought of as the catalog of installed protocols. Applications can retrieve the contents of this catalog (for more information, see [WSAEnumProtocols](#), [WSCEnumProtocols](#), and [WSCEnumProtocols32](#)), and by examining the available WSAPROTOCOL\_INFO structures, discover the communications attributes associated with each protocol.

## Layered Protocols and Protocol Chains in the SPI

Windows Sockets 2 accommodates 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 such a layered protocol would be a security layer that adds protocol to the connection establishment process in order to perform authentication and to establish a mutually agreed upon encryption scheme. Such a security protocol would generally require the services of an underlying reliable transport protocol such as TCP or SPX. The term base protocol refers to a protocol such as TCP or SPX which is fully capable of performing data communications with a remote endpoint, and the term layered protocol is used to describe a protocol that cannot stand alone. A protocol chain would then be defined as one or more layered protocols strung together and anchored by a base protocol.

This stringing of layered protocols and base protocols into chains can be accomplished by arranging for the layered protocols to support the Winsock SPI at both their upper and lower edges. A special [WSAPROTOCOL\\_INFO](#) structure is created which refers to the protocol chain as a whole, and which describes the explicit order in which the layered protocols are joined. This is illustrated in the following graphic.



# Transport Division of Responsibilities Between DLL and Service Providers

3/5/2021 • 4 minutes to read • [Edit Online](#)

This section provides an overview of the division of responsibility between the Ws2\_32.dll and transport service providers.

## Ws2\_32.dll Functionality for Transport

The major task of the data transport portion of the Ws2\_32.dll is to serve as traffic manager between service providers and applications. With several different service providers interacting with the same application, each service provider interacts strictly with the Ws2\_32.dll. The DLL takes care of:

- Selecting an appropriate service provider for creating sockets based on a protocol description.
- Forwarding application procedure calls involving a socket to the appropriate service provider that created or controls that socket. Service providers are unaware that any of this is happening.

They do not need to be concerned about the details of cooperating with one another or even the existence of other service providers. By abstracting the service providers into a consistent DLL interface and performing this automatic traffic routing function, the Ws2\_32.dll allows applications to interact with a variety of providers without requiring the applications to be aware of the divisions between providers, where different providers are installed.

The Ws2\_32.dll relies on the parameters of the API socket creation functions ([socket](#) and [WSASocket](#)) to determine which service provider to utilize. The selected transport service provider will be invoked through the [WSPSocket](#) function. In the case of the [socket](#) function, the Ws2\_32.dll finds the first entry in the set of installed [WSAPROTOCOL\\_INFO](#) structures that matches the values supplied in the tuple formed by the *(address family, socket type, protocol)* parameters. To preserve backward compatibility, the Ws2\_32.dll treats the value of zero for either *address family* or *socket type* as a wildcard value. The value of zero for protocol is not considered a wildcard value by the Ws2\_32.dll unless such behavior is indicated for a particular protocol by having the PFL\_MATCHES\_PROTOCOL\_ZERO flag set in the [WSAPROTOCOL\\_INFO](#) structure.

For the [WSASocket](#) function, if null is supplied for *lpProtocolInfo*, the behavior is exactly as just described for [socket](#). If a [WSAPROTOCOL\\_INFO](#) structure is referenced, however, the Ws2\_32.dll does not perform any matching function but immediately relays the socket creation request to the transport service provider associated with the indicated [WSAPROTOCOL\\_INFO](#) structure. The values for the *(address family, socket type)* tuple are supplied intact to the service provider in the [WSPSocket](#) function. Service providers are free to ignore or pay attention to the values of the *(address family, socket type)* parameters as is appropriate, but they must not indicate an error condition when the value of either *address family* or *socket type* is zero. In addition, service providers must not indicate an error condition when the manifest constant FROM\_PROTOCOL\_INFO is contained in any of the *(address family, socket type)* parameters. This value simply indicates that the application wishes to use the values found in the corresponding parameters of the [WSAPROTOCOL\\_INFO](#) structure: (*iAddressFamily*, *iSocketType*, *iProtocol*).

As part of socket creation, a service provider informs the Ws2\_32.dll about the association between itself and the new socket by means of parameters passed to [WPUCreateSocketHandle](#) or [WPUModifyIFSHandle](#). The Ws2\_32.dll keeps track of this association between socket handles and particular service providers. Whenever an application interface function that refers to a socket handle is called, the Ws2\_32.dll looks up the association and calls the corresponding service provider interface function of the appropriate service provider.

In addition to its major traffic routing service, the Ws2\_32.dll provides a number of other services such as protocol enumeration, socket descriptor management (allocation, deallocation, and context value association) for nonfile-system service providers, blocking hook management on a per-thread basis, byte-swapping utilities, queuing of asynchronous procedure calls (APCs) to facilitate invocation of I/O completion routines, and version negotiation between applications and the Ws2\_32.dll, as well as between the Ws2\_32.dll and service providers.

## Transport Service Provider Functionality

Service providers implement the actual transport protocol which includes such functions as setting up connections, transferring data, exercising flow control and error control, etc. The Ws2\_32.dll has no knowledge about how requests to service providers are realized; this is up to the service provider implementation. The implementation of such functions may differ greatly from one provider to another. Service providers hide the implementation-specific details of how network operations are accomplished.

Transport service providers can be broadly divided into two categories: those whose socket descriptors are real file system handles (and are hereafter referred to as Installable File System (IFS) providers; the remainder are referred to as non-IFS providers. The Ws2\_32.dll always passes the transport service provider's socket descriptor on up to the Windows Sockets application, so applications are free to take advantage of socket descriptors that are file system handles if they so choose.

To summarize: service providers implement the low-level network-specific protocols. The Ws2\_32.dll provides the medium-level traffic management that interconnects these transport protocols with applications.

Applications in turn provide the policy of how these traffic streams and network-specific operations are used to accomplish the functions desired by the user.

# Transport Mapping Between API and SPI Functions

3/5/2021 • 2 minutes to read • [Edit Online](#)

The Winsock Transport SPI is similar to the Winsock API in that all of the basic socket functions appear. When a new version of a Winsock function and the original version both exist in the API, only the new version will show up in the SPI. This is illustrated in the following list.

- [connect](#) and [WSAConnect](#) both map to [WSPConnect](#)
- [accept](#) and [WSAAccept](#) map to [WSPAccept](#)
- [socket](#) and [WSASocket](#) map to [WSPSocket](#)

Other API functions that are collapsed into the enhanced versions in SPI include:

- [send](#)
- [sendto](#)
- [recv](#)
- [recvfrom](#)
- [ioctlsocket](#)

Support functions like [htonl](#), [htons](#), [ntohl](#), and [ntohs](#) are implemented in the Ws2\_32.dll, and are not passed down to service providers. The same holds true for the WSA versions of these functions.

Windows Sockets service provider enumeration and the blocking hook-related functions are realized in the Ws2\_32.dll, thus [WSAEnumProtocols](#), [WSAIsBlocking](#), [WSASetBlockingHook](#), and [WSAUUnhookBlockingHook](#) do not appear as SPI functions.

Because error codes are returned along with SPI functions, equivalents of [WSAGetLastError](#) and [WSASetLastError](#) are not required in the SPI.

The event object manipulation and wait functions, including [WSACreateEvent](#), [WSACloseEvent](#), [WSASetEvent](#), [WSAResetEvent](#), and [WSAWaitForMultipleEvents](#) are mapped directly to native Windows services and thus are not present in the SPI.

All the TCP/IP-specific name conversion and resolution functions in Windows Sockets 1.1 such as [GetXbyY](#), [WSAAAsyncGetXByY](#), and [WSACancelAsyncRequest](#), as well as [gethostname](#) are implemented within the Ws2\_32.dll in terms of the new name resolution facilities. For more information, see [Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 SPI](#). Conversion functions such as [inet\\_addr](#) and [inet\\_ntoa](#) are implemented within the Ws2\_32.dll.

# Function Extension Mechanism in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

Because the Winsock DLL itself is no longer supplied by each individual stack vendor, it is not possible for a stack vendor to offer extended functionality by adding entry points to the Winsock DLL. To overcome this limitation, Winsock takes advantage of the new **WSAIoctl** function to accommodate service providers who wish to offer provider-specific functionality extensions. This mechanism presupposes 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.

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 and the output buffer will contain 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 facilitates common and/or popular extension functions to be offered by multiple service providers. 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.

# Transport Configuration and Installation

3/5/2021 • 2 minutes to read • [Edit Online](#)

In order for a transport protocol to be accessible through Windows Sockets it must be properly installed on the system and registered with Windows Sockets. When a transport service provider is installed by invoking a vendor's installation program, configuration information must be added to a configuration database to give the Ws2\_32.dll required information regarding the service provider. The Ws2\_32.dll exports several installation functions, [WSCInstallProvider](#) and [WSCInstallProviderAndChains](#), for the vendor's installation program to supply the relevant information about the service provider to be installed. This information includes, for example, the name and path to the service provider DLL and a list of [WSAPROTOCOL\\_INFO](#) structures that this provider can support. The Ws2\_32.dll also provides a function, [WSCDeinstallProvider](#), for a vendor's deinstallation program to remove all the relevant information from the configuration database maintained by the Ws2\_32.dll. The exact location and format of this configuration information is private to the Ws2\_32.dll, and can only be manipulated by the above-mentioned functions.

On 64-bit platforms, there are similar functions that operate on a 32-bit and 64-bit catalogs. These functions include [WSCInstallProvider64\\_32](#), [WSCInstallProviderAndChains64\\_32](#), and [WSCDeinstallProvider32](#).

The order in which transport service providers are initially installed governs the order in which they are enumerated through [WSCEnumProtocols](#) and [WSCEnumProtocols32](#) at the service provider interface, or through [WSAEnumProtocols](#) at the application interface. More importantly, this order also governs the order in which protocols and service providers are considered when a client requests creation of a socket based on its address family, type, and protocol identifier. Windows Sockets 2 includes an applet called Sporder.exe that allows the catalog of installed protocols to be reordered interactively after protocols have already been installed. Windows Sockets 2 also includes an auxiliary DLL, Sporder.dll, that exports a procedural interface for reordering protocols. This procedural interface consists of a single procedure called [WSCWriteProviderOrder](#).

## Installing Layered Protocols and Protocol Chains

The [WSAPROTOCOL\\_INFO](#) structure supplied with each protocol to be installed indicates whether the protocol is a base protocol, layered protocol, or protocol chain. The value of the *ProtocolChain.ChainLen* parameter is interpreted as shown in the following table.

VALUE	MEANING
0	Layered protocol.
1	Base protocol (or chain with only one component).
>1	Protocol chain.

Installation of protocol chains can only occur after successful installation of all of the constituent components (base protocols and layered protocols). The [WSAPROTOCOL\\_INFO](#) structure for a protocol chain uses the *ProtocolChain* parameter to describe the length of the chain and the identity of each component. The individual protocols that make up a chain are listed in order in the *ProtocolChain.ChainEntries* array, with the zeroth element of the array corresponding to the first layered provider. Protocols are identified by their *CatalogEntryID* values, which are assigned by the Ws2\_32.dll at protocol installation time, and can be found in the

**WSAPROTOCOL\_INFO** structure for each protocol.

The values for the remaining parameters in the protocol chain's **WSAPROTOCOL\_INFO** structure should be chosen to reflect the attributes and identifiers that best characterize the protocol chain as a whole. When selecting these values, developers should bear in mind that communications over protocol chains can only occur when both endpoints have compatible protocol chains installed, and that applications must be able to recognize the corresponding **WSAPROTOCOL\_INFO** structure.

When a base protocol is installed, it is not necessary to make any entries in the *ProtocolChain.ChainEntries* array. It is implicitly understood that the sole component of this chain is already identified in the *CatalogEntryID* parameter of the same **WSAPROTOCOL\_INFO** structure. Also note that protocol chains may not include multiple instances of the same layered protocol.

# Winsock Transport Service Provider Requirements

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following sections provide a description of each of the functional areas which transport service providers are required to implement. Where appropriate, implementation considerations and guidelines are also provided.

- [Service Provider Activation](#)
- [Socket Creation and Descriptor Management](#)
- [Blocking Operations](#)
- [Event Objects in the Windows Sockets 2 SPI](#)
- [Notification of Network Events](#)
- [Socket Groups in the Windows Sockets 2 SPI](#)
- [Quality of Service in the Windows Sockets 2 SPI](#)
- [Socket Connections on Connection-Oriented Protocols](#)
- [Socket Connections on Connectionless Protocols](#)
- [Socket I/O](#)
- [Shared Sockets in the SPI](#)
- [Protocol-Independent Multicast and Multipoint in the SPI](#)
- [Socket Options and IOCTLs](#)
- [Summary of SPI Functions](#)

# Service Provider Activation

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following sections describe the sequence of events involved in bringing a transport service provider DLL into memory, initializing it, and eventually, uninitializing it.

- [Initialization](#)
- [Cleanup](#)
- [Error Reporting and Parameter Validation](#)
- [Byte Ordering Assumptions](#)

# Initialization

3/5/2021 • 3 minutes to read • [Edit Online](#)

The *Ws2\_32.dll* loads the service provider's interface DLL into the system by using the standard Microsoft Windows dynamic library loading mechanisms, and initializes it by calling **WSPStartup**. This is typically triggered by an application calling either **socket** or **WSASocket** to create a new socket to be associated with a service provider whose interface DLL is not currently loaded into memory. The path to each service provider's interface DLL is stored by the *Ws2\_32.dll* at the time the service provider is being installed. See [Installation and Configuration Functions](#) for more information.

Over time, different versions may exist for the Winsock DLLs, applications, and service providers. New versions may define new features and new parameters to data structures and bit parameters, etc. Version numbers therefore indicate how to interpret various data structures.

To allow optimal mixing and matching of different versions of applications, versions of the *Ws2\_32.dll* itself, and versions of service providers by different vendors, the SPI provides a version negotiation mechanism for use between the *Ws2\_32.dll* and the service providers. This version negotiation is handled by **WSPStartup**. Basically, the *Ws2\_32.dll* passes to the service provider the highest version numbers with which it is compatible. The service provider compares this with its own supported range of version numbers. If these ranges overlap, the service provider returns a value within the overlapping portion of the range as the result of the negotiation. Usually, this should be the highest possible value. If the ranges do not overlap, the two parties are incompatible and the function returns an error.

**WSPStartup** must be called at least once by each client process, and may be called multiple times by *Ws2\_32.dll* or other entities. A matching **WSPCleanup** must be called for each successful **WSPStartup** call. The service provider should maintain a reference count on a per-process basis. On each **WSPStartup** call, the caller may specify any version number supported by the SP DLL.

A service provider must store the pointer to the client's upcall dispatch table that is received as a **WSPStartup** parameter on a per-process basis. If a given process calls **WSPStartup** multiple times, the service provider must use only the most recently supplied dispatch table pointer.

As part of the service provider initialization process The *Ws2\_32.dll* retrieves the service provider's dispatch table through the *lpProcTable* parameter in order to obtain entry points to the rest of the SPI functions specified in this document.

Using a dispatch table (as opposed to the usual DLL mechanisms for accessing entry points) serves two purposes:

- It is more convenient for the *Ws2\_32.dll* since a single call can be made to discover the entire set of entry points.
- It enables layered service providers formed into protocol chains to operate more efficiently.

## Initializing Protocol Chains

At the time the **WSAPROTOCOL\_INFO** structure for a protocol chain is installed, the path to the first layered provider in the chain is also specified. When a protocol chain is initialized, the *Ws2\_32.dll* uses this path to load the provider DLL and then invokes **WSPStartup**. Since **WSPStartup** includes a pointer to the chain's **WSAPROTOCOL\_INFO** structure as one of its parameters, layered providers can determine what type of chain they are being initialized into, and the identity of the next lower layer in the chain. A layered provider would then in turn load the next protocol provider in the chain and initialize it with a call to **WSPStartup**, and so forth. Whenever the next lower layer is another layered provider, the chain's **WSAPROTOCOL\_INFO** structure must

be referenced in the **WSPStartup** call. When the next lower layer is a base protocol (signifying the end of the chain), the chain's **WSAPROTOCOL\_INFO** structure is no longer propagated downward. Instead, the current layer must reference a **WSAPROTOCOL\_INFO** structure that corresponds to the protocol that the base provider should use. Thus, the base provider has no notion of being involved in a protocol chain.

The dispatch table provided by any given layered provider will, in many instances, duplicate the entry points of an underlying provider. The layered provider would only insert its own entry points for functions that it needed to be directly involved in. Note, however, that it is imperative that a layered provider not modify the contents of the upcall table that it received when calling **WSPStartup** on the next lower layer in a protocol chain. These upcalls must be made directly to the Windows Sockets 2 DLL.

# Cleanup

3/5/2021 • 2 minutes to read • [Edit Online](#)

The Ws2\_32.dll (and layered protocols) will call **WSPCleanup** once for each invocation of **WSPStartup**. On each invocation, **WSPCleanup** should decrement a per-process reference counter, and when the counter reaches zero, the service provider must prepare itself to be unloaded from memory. The first order of business is to finish transmitting any unsent data on sockets that are configured for a graceful close. Thereafter, any and all resources held by the provider are to be freed. The service provider must be left in a state where it can be immediately reinitialized by a call to **WSPStartup**.

# Error Reporting and Parameter Validation

3/5/2021 • 2 minutes to read • [Edit Online](#)

The scheme for error reporting differs between the SPI and API interfaces. Windows Sockets service providers report errors along with the function returning, as opposed to the per-thread based approach utilized in the API. The Ws2\_32.dll uses the service provider's per-function error code to update the per-thread error value that is obtained through the [WSAGetLastError](#) API function. Service providers are still required, however, to maintain the per-socket based error which can be retrieved through the SO\_ERROR socket option.

The Ws2\_32.dll performs parameter validation only on function calls that are implemented entirely within itself. Service providers are responsible for performing all of their own parameter validation.

# Byte Ordering Assumptions

3/5/2021 • 2 minutes to read • [Edit Online](#)

A service provider should treat all **sockaddr** components exclusive of the address family parameter as if they are in the network byte order, and the address family parameter as in the local computer's byte order. It is the Winsock application's responsibility to make sure that addresses contained in **sockaddr** structures are properly arranged. The Winsock API provides a number of conversion routines to simplify this task. Currently these routines understand conversion between the local host's natural byte order and either big-Endian or little-Endian network byte ordering. The Winsock architecture can support other byte ordering schemes in the future.

# Socket Creation and Descriptor Management

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following sections describe aspects of creating new sockets and the allocation of socket descriptors.

- [Descriptor Allocation](#)
- [Socket Attribute Flags and Modes](#)
- [Closing Sockets](#)

# Descriptor Allocation

3/5/2021 • 2 minutes to read • [Edit Online](#)

While Windows Sockets service providers are encouraged to implement sockets as installable file system (IFS) objects, the Winsock architecture also accommodates service providers whose socket handles are not IFS objects. Providers with IFS handles indicate this through the XP1\_IFS\_HANDLES attribute bit in the [WSAPROTOCOL\\_INFO](#) structure. (Note: the XP1\_IFS\_HANDLES attribute bit was not included in release 2.0.8 of the API specification, but has since been added through the errata mechanism.) Winsock SPI clients may take advantage of providers whose socket descriptors are IFS handles by using these descriptors with standard Windows I/O facilities, such as [ReadFile](#) and [WriteFile](#).

Whenever an IFS provider creates a new socket descriptor, it is mandatory that the provider call [WPUModifyIFSHandle](#) prior to supplying the new handle to a Windows Sockets SPI client. This function takes a provider identifier and a proposed IFS handle from the provider as input and returns a (possibly) modified handle. The IFS provider must supply only the modified handle to its client, and all requests from the client will reference only this modified handle. The modified handle is guaranteed to be indistinguishable from the proposed handle as far as the operating system is concerned. Thus in most instances, the service provider will simply choose to use only the modified handle in all of its internal processing. The purpose of this modification function is to allow the Ws2\_32.dll to greatly streamline the process of identifying the service provider associated with a given socket.

Providers that do not return IFS handles must obtain a valid handle from the Ws2\_32.dll via the [WPUCreateSocketHandle](#) call. The nonIFS provider must offer only a Windows Sockets 2.DLL-supplied handle to its client, and all requests from the client will reference only these handles. As a convenience to service provider implementers, one of the input parameters supplied by a provider in [WPUCreateSocketHandle](#) is a DWORD context value. The Ws2\_32.dll associates this context value with the allocated socket handle and allows a service provider to retrieve the context value at any time through the [WPUQuerySocketHandleContext](#) call. A typical use for this context value would be to store a pointer to a service provider maintained data structure used to store socket state information.

# Socket Attribute Flags and Modes

3/5/2021 • 2 minutes to read • [Edit Online](#)

There are several socket attributes which can be indicated through the *flags* parameter in [WSPSocket](#). The WSA\_FLAG\_OVERLAPPED flag indicates that a socket will be used for overlapped I/O operations. Support of this attribute is mandatory for all service providers. See [Overlapped I/O](#) for more information. Note that creating a socket with the overlapped attribute has no impact on whether a socket is currently in blocking or nonblocking mode. Sockets created with the overlapped attribute may be used to perform overlapped I/O, and doing so does not change the blocking mode of a socket. Since overlapped I/O operations do not block, the blocking mode of a socket is irrelevant for these operations.

Four additional attribute flags are used when creating sockets that are to be used for multipoint and/or multicast operations, and support for these attributes is optional. Providers that support multipoint attributes indicate this through the XP1\_SUPPORT\_MULTIPOINT bit in their respective [WSAPROTOCOL\\_INFO](#) structures. See [WSPSocket](#) and [Protocol-Independent Multicast and Multipoint in the API](#) for the definition and usage of each of these flags. Only sockets that are created with multipoint-related attributes can be used in the [WSPJoinLeaf](#) function for creating multipoint sessions.

A socket is in one of two modes, blocking and nonblocking, at any time. Sockets are created in the blocking mode by default, and can be changed to nonblocking mode by calling [WSPAsyncSelect](#), [WSPEventSelect](#), or [WSPIoctl](#). A socket can be switched back to blocking mode by using [WSPIoctl](#) if no [WSPAsyncSelect](#) or [WSPEventSelect](#) is active. The mode for a socket only affects those functions which may block, and does not affect overlapped I/O operations. See [Blocking Operations](#) for more information.

# Closing Sockets

3/5/2021 • 2 minutes to read • [Edit Online](#)

**WSPCloseSocket** releases the socket descriptor so that any pending operations in any threads of the same process will be aborted, and any further reference to it will fail. Note that a reference count should be employed for shared sockets, and only if this is the last reference to an underlying socket, should the information associated with this socket be discarded, provided graceful close is not requested (that is, SO\_DONTLINGER is not set). In the case of SO\_DONTLINGER being set, any data queued for transmission will be sent, if possible, before information associated with the socket is released. See **WSPCloseSocket** for more information.

For nonIFS service providers, **WPUCloseSocketHandle** must be invoked to release the socket handle back to the Windows Sockets 2 DLL.

# Blocking Operations

3/5/2021 • 2 minutes to read • [Edit Online](#)

The notion of blocking in a Windows environment has historically been a very important one. In Windows Sockets 1.1 environments, blocking calls were discouraged since they tended to disable ongoing interaction with the system. Additionally, they employ a pseudoblocking technique which, for a variety of reasons, does not always work as intended. However, in preemptively scheduled Windows environments, blocking calls make much more sense, can be implemented by native operating system services, and are in fact a generally preferred mechanism. The Winsock 2 API no longer supports psuedoblocking, but because the Windows Sockets 1.1 compatibility shims must continue to emulate this behavior, service providers must support this as described in the following.

# Pseudo Blocking and True Blocking

3/5/2021 • 2 minutes to read • [Edit Online](#)

In 16 Windows environments, true blocking is not supported by the operating system; therefore, a blocking operation that cannot be completed immediately is handled as follows:

- The service provider initiates the operation, and then enters a loop during which it dispatches any Windows messages (yielding the processor to another thread if necessary)
- It then checks for the completion of the Windows Sockets function.
- If the function has completed, or if **WSPCancelBlockingCall** has been invoked, the loop is terminated and the blocking function completes with an appropriate result.

This is what is meant by the term pseudo blocking, and the loop referred to above is known as the default blocking hook.

# Blocking Hook

3/5/2021 • 2 minutes to read • [Edit Online](#)

Although this mechanism is sufficient for simple applications, it cannot support the complex message-dispatching requirements of more advanced applications such as those using the Multiple Document Interface (MDI) model. For such applications, a thread-specific blocking hook may be installed by the application. This will be called by the service provider instead of the default blocking hook described in the preceding. A service provider must retrieve a pointer to the per-thread blocking hook from the `Ws2_32.dll` by calling [WPUQueryBlockingCallback](#). If the application has not installed its own blocking hook a pointer to the default blocking hook function will be returned.

A Windows Sockets service provider cannot assume that an application-supplied blocking hook allows message processing to continue as the default blocking hook does. Some applications cannot tolerate the possibility of reentrant messages while a blocking operation is outstanding. Such an application's blocking hook function would simply return `FALSE`. If a service provider depends on messages for its internal operation, it may execute `PeekMessage(hMyWnd...)` before executing the application's blocking hook so that it can get its own messages without affecting the rest of the system.

There is no default blocking hook installed in preemptive multithreaded versions of Windows. This is because other processes will not be blocked if a single application is waiting for an operation to complete (and hence not calling `PeekMessage` or `GetMessage` which causes the application to yield the processor in nonpreemptive Windows). When the service provider calls [WPUQueryBlockingCallback](#) a null pointer will be returned indicating that the provider is to use native operating system blocking functions. However, in order to preserve backward compatibility, an application-supplied blocking hook can still be installed on a per-thread basis in Windows.

The Winsock service provider calls the blocking hook only if all of the following are true: the routine is one which is defined as being able to block, the specified socket is a blocking socket, and the request cannot be completed immediately. If only nonblocking sockets and [WSPAsyncSelect/WSPEventSelect](#) instead of [WSPSelect](#) are used, then the blocking hook will never be called.

## NOTE

If, during the time pseudoblocking is being used to block a thread, a Windows message is received for the thread, there is a risk that the thread will attempt to issue another Winsock call. Because of the difficulty of managing this condition safely, the Windows Sockets 1.1 specification disallowed this behavior. It is not permissible for a given thread to make multiple, nested Winsock function calls. Only one outstanding function call is allowed for a particular thread. Any nested Winsock function calls fail with the error `WSAEINPROGRESS`. It should be emphasized that this restriction applies to both blocking and nonblocking operations, but only in Windows Sockets 1.1 environments. There are a few exceptions to this rule, including two functions that allow an application to determine whether a pseudoblocking operation is in fact in progress, and to cancel such an operation if need be. These are described in the following.

# Cancelling Blocking Operations

3/5/2021 • 2 minutes to read • [Edit Online](#)

A thread may, at any time, call [WSAIIsBlocking](#) to determine whether or not a blocking call is currently in progress. (This function is implemented within the Windows Sockets 1.1 compatibility shims and hence has no SPI counterpart.) Clearly this is only possible when pseudo blocking, as opposed to true blocking, is being employed by the service provider. When necessary, [WSPCancelBlockingCall](#) may be called at any time to cancel any current pseudo blocking operation.

# Event Objects in the Windows Sockets 2 SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

Event objects are introduced in Windows Sockets 2 as a general synchronization mechanism between Winsock service providers and applications. They are used for a number of purposes that include indicating the completion of overlapped operations and the occurrence of one or more network events.

# Creating Event Objects

3/5/2021 • 2 minutes to read • [Edit Online](#)

The Ws2\_32.dll provides facilities for event object creation to both applications and service providers, although in most instances event objects will be created by applications. Event object services are made available to Windows Sockets service providers through [WPUCreateEvent](#) simply as a convenience mechanism for any internal processing that may benefit from same. Note that the event object handle is only valid in the context of the calling process. In Windows environments the realization of event objects is through the native event services provided by the operating system.

# Using Event Objects (Windows Sockets 2)

3/5/2021 • 2 minutes to read • [Edit Online](#)

Windows Sockets event objects are simple constructs that can be created and closed, set and cleared, waited upon and polled. The accepted model is for clients to create an event object and supply the handle as a parameter (or as a component of a parameter structure) in functions such as [WSPSend](#) and [WSPEventSelect](#). When the nominated condition has occurred, the service provider uses the handle to set the event object by calling [WPUSetEvent](#). Meanwhile, the Winsock SPI client may either block and wait or poll until the event object becomes set (signaled). The client may subsequently clear or reset the event object and use it again.

# Destroying Event Objects

3/5/2021 • 2 minutes to read • [Edit Online](#)

The entity that creates an event object (application or service provider) is responsible for destroying it when it is no longer required. Service providers do this through **WPUCloseEvent**.

# Notification of Network Events

3/5/2021 • 2 minutes to read • [Edit Online](#)

One of the most important responsibilities of a data transport service provider is that of providing indications to the client when certain network events have occurred. The list of defined network events consists of the following:

- **FD\_CONNECT**— A connection to a remote host or to a multicast session has been completed.
- **FD\_ACCEPT**— A remote host is making a connection request.
- **FD\_READ**— Network data has arrived which is available to be read.
- **FD\_WRITE**— Space has become available in the service provider's buffers so that additional data may now be sent.
- **FD\_OOB**— Out of band data is available to be read.
- **FD\_CLOSE**— The remote host has closed down the connection.
- **FD\_QOS**— A change has occurred in negotiated QoS levels.
- **FD\_GROUP\_QOS**— Reserved.
- **FD\_ROUTING\_INTERFACE\_CHANGE**— A local interface that should be used to reach the destination specified in SIO\_ROUTING\_INTERFACE\_CHANGE IOCTL has changed.
- **FD\_ADDRESS\_LIST\_CHANGE**— The list of local addresses to which application can bind has changed.

The set of network events enumerated above is sometimes referred to as the **FD\_XXX** events. Indication of the occurrence of one or more of such network events may be given in a number of ways depending on how the client requests notification.

# Selects

3/5/2021 • 2 minutes to read • [Edit Online](#)

In the original Berkeley Software Distribution (BSD) socket interface the **select** function was the standard (and only) means to obtain network event indications. For each socket, information on read, write, or error status can be polled or waited on. See [WSPSelect](#) for more information. Note that network event FD\_QOS cannot be obtained by this approach.

# Windows Messages

3/5/2021 • 2 minutes to read • [Edit Online](#)

Windows Sockets 1.1 introduced the async-select mechanism to provide network event indications that did not involve either polling or blocking. The [WSPAsyncSelect](#) function is used to register an interest in one or more network events as listed in the preceding, and supply a window handle to be used for notification. When a nominated network event occurs, a client-specified Windows message is sent to the indicated window. The service provider must use the [WPUPostMessage](#) function to accomplish this.

In Windows, this may not be the most efficient notification mechanism, and is inconvenient for daemons and services that do not want to open any windows. The event object signaling mechanism described in the following is introduced to solve this problem.

# Event Object Signaling

3/5/2021 • 2 minutes to read • [Edit Online](#)

**WSPEventSelect** behaves exactly like **WSPAsyncSelect** except that, rather than cause a Windows message to be sent upon the occurrence of any nominated FD\_XXX network event (for example, FD\_READ, FD\_WRITE, etc.), a designated event object is set.

Also, the fact that a particular FD\_XXX network event has occurred is remembered by the service provider. This is needed since the occurrence of any and all nominated network events will result in a single event object becoming signaled. A call to **WSPEnumNetworkEvents** causes the current contents of the network event memory to be copied to a client-supplied buffer and the network event memory to be cleared. The client may also designate a particular event object to be cleared atomically along with the network event memory.

# Socket Groups in the Windows Sockets 2 SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

All use of socket groups is reserved.

# Socket Group Operations

3/5/2021 • 2 minutes to read • [Edit Online](#)

All use of socket groups is reserved.

# Required Socket Grouping Behavior

3/5/2021 • 2 minutes to read • [Edit Online](#)

All use of socket groups is reserved.

# Recommended Socket Grouping Behavior

3/5/2021 • 2 minutes to read • [Edit Online](#)

All use of socket groups is reserved.

# Quality of Service in the Windows Sockets 2 SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

Quality of Service (QoS) is implemented in Windows through various QoS components supported on Windows Server 2003, Windows XP, and Windows 2000. A detailed explanation of QoS and its components is located in a section dedicated to Quality of Service, located in the Microsoft Windows Software Development Kit (SDK). Each QoS component, and the role it plays in the QoS implementation, is explained in this QoS section, with additional guidance provided for the implementation of QoS-enabled applications and services. Please see the [Quality of Service \(QoS\)](#) section of the Windows SDK for more detailed information.

This section describes Quality of Service capabilities available to Winsock developers. The following list describes the topics in this section:

- [Usage Model](#)
- [QoS Updates](#)
- [Default Values in the SPI](#)
- [QoS Templates in the SPI](#)

## Related topics

[Quality of Service \(QoS\)](#)

[What Is QoS?](#)

[Winsock ATM QoS Extension](#)

[Winsock IOCTLs](#)

# Usage Model

3/5/2021 • 2 minutes to read • [Edit Online](#)

Usage and implementation guidance for QoS-enabled applications and services are provided in a separate section in the Microsoft Windows Software Development Kit (SDK) that is dedicated to QoS. Please see the [Quality of Service \(QoS\)](#) section of the Windows SDK for more detailed information on the QoS usage model.

## Related topics

[Quality of Service \(QoS\)](#)

[Quality of Service in the Windows Sockets 2 SPI](#)

# QoS Updates

3/5/2021 • 2 minutes to read • [Edit Online](#)

The Quality of Service (QoS) implementation details are provided in a separate, dedicated QoS section in the Microsoft Windows Software Development Kit (SDK). Please see the [Quality of Service](#) section of the Windows SDK for information on QoS updates.

## Related topics

[Quality of Service \(QoS\)](#)

[Quality of Service in the Windows Sockets 2 SPI](#)

# Default Values in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

The Quality of Service (QoS) implementation details are provided in a separate, dedicated QoS section in the Microsoft Windows Software Development Kit (SDK). Please see the [Quality of Service](#) section of the Windows SDK for information on QoS usage guidelines regarding default values.

## Related topics

[Quality of Service \(QoS\)](#)

[Quality of Service in the Windows Sockets 2 SPI](#)

# QoS Templates in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

The Quality of Service (QoS) implementation details are provided in a separate, dedicated QoS section in the Microsoft Windows Software Development Kit (SDK). Please see the [Quality of Service](#) section of the Windows SDK for information on QoS usage guidelines, including available QoS templates and their associated [FLOWSPEC](#) values.

## Related topics

[Quality of Service \(QoS\)](#)

[Quality of Service in the Windows Sockets 2 SPI](#)

[FLOWSPEC](#)

# Socket Connections on Connection-Oriented Protocols

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following paragraphs describe the semantics applicable to socket connections over connection-oriented protocols:

- [Binding to a Local Address](#)
- [Protocol Basics: Listen, Connect, Accept](#)
- [Determining Local and Remote Names](#)
- [Enhanced Functionality at Connect Time](#)
- [Connection Shutdown](#)

# Binding to a Local Address

3/5/2021 • 2 minutes to read • [Edit Online](#)

Before a socket can be used to set up a connection or receive a connection request, it needs to be bound to a local address. This could be done explicitly by calling [WSPPBind](#), or implicitly by [WSPPConnect](#) if the socket is unbound when this function is called.

# Protocol Basics: Listen, Connect, Accept

3/5/2021 • 2 minutes to read • [Edit Online](#)

The basic operations involved with establishing a socket connection can be most conveniently explained in terms of the client-server paradigm.

The server side will first create a socket, bind it to a well known local address (so that the client can find it), and put the socket in listening mode, through [WSPListen](#), in order to prepare for any incoming connection requests and to specify the length of the connection backlog queue. Service providers hold pending connection requests in a backlog queue until they are acted upon by the server or are withdrawn (due to a time-out) by the client. A service provider may silently ignore requests to extend the size of the backlog queue beyond a provider-defined upper limit.

At this point, if a blocking socket is being used, the server side may immediately call [WSPAccept](#) which will block until a connection request is pending. Conversely, the server may also use one of the network event indication mechanisms discussed previously to arrange for notification of incoming connection requests.

Depending on the notification mechanism selected, the provider will either issue a Windows message or signal an event object when connection requests arrive. See [Notification of Network Events](#) for how to register for the FD\_ACCEPT network event.

The client side will create an appropriate socket, and initiate the connection by calling [WSPConnect](#), specifying the known server address. Clients usually do not perform an explicit [bind](#) operation prior to initiating a connection, allowing the service provider to perform an implicit bind on their behalf. If the socket is in blocking mode, [WSPConnect](#) will block until the server has received and acted upon the connection request (or until a time-out occurs). Otherwise, the client should use one of the network event indication mechanisms discussed previously to arrange for notification of a new connection being established. Depending on the notification mechanism selected, the provider will indicate this either through a Windows message or by signaling an event object.

When the server side invokes [WSPAccept](#), the service provider calls the application-supplied condition function, using function parameters to pass into the server information from the top entry in the connection request backlog queue. This information includes such things as address of connecting host, any available user data, and any available QoS information. Using this information the server's condition function determines whether to accept the request, reject the request, or defer making a decision until later. This decision is indicated through the return value of the condition function. See [Notification of Network Events](#) for how to register for the FD\_CONNECT network event.

If the server decides to accept the request, the provider must create a new socket with all of the same attributes and event registrations as the listening socket. The original socket remains in the listening state so that subsequent connection requests can be received. Through output parameters of the condition function, the server may also supply any response user data and assign QoS parameters (assuming that these operations are supported by the service provider).

# Determining Local and Remote Names

3/5/2021 • 2 minutes to read • [Edit Online](#)

**WSPGetSockName** is used to retrieve the local address for bound sockets. This is especially useful when a **WSPConnect** call has been made without doing a **WSPBind** first; **WSPGetSockName** provides the only means to determine the local association which has been set implicitly by the provider.

After a connection has been set up, **WSPGetPeerName** can be used to determine the address of the peer to which a socket is connected.

# Enhanced Functionality at Connect Time

3/5/2021 • 2 minutes to read • [Edit Online](#)

Windows Sockets 2 offers an expanded set of operations that can occur coincident to establishing a socket connection. The service provider requirements for implementing these features are described below.

## Conditional Acceptance

As described previously, [WSPAccept](#) invokes a client-supplied condition function that uses input parameters to supply information about the pending connection request. This information can be used by the client to accept or reject a connection request based on caller information such as caller identifier, QoS, etc. If the condition function returns CF\_ACCEPT, a new socket is created with the same properties as the listening socket, and a handle to the new socket is returned. If the condition function returns CF\_REJECT, the connection request should be rejected. If the condition function returns CF\_DEFER, the accept/reject decision cannot be made immediately, and the service provider must leave the connection request on the backlog queue. The client must call [WSPAccept](#) again, when it is ready to make a decision, and arrange for the condition function to return either CF\_ACCEPT or CF\_REJECT. While a deferred connection request is at the top of the backlog queue, the service provider does not issue any further indications for pending connection requests.

## Exchanging User Data at Connect Time

Some protocols allow a small amount of user data to be exchanged at connect time. If such data has been received from the connecting host, it is placed in a service provider buffer, and a pointer to this buffer along with a length value are supplied to the Winsock SPI client through input parameters to the [WSPAccept](#) condition function. If the Winsock SPI client has response data to return to the connecting host, it may copy this into a buffer that is supplied by the service provider. A pointer to this buffer and an integer indicating buffer size are also supplied as condition function input parameters (if supported by the protocol).

## Establishing Socket Groups

All use of socket groups is reserved.

# Connection Shutdown

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following describes operations incident to shutting down an established socket connection.

## Initiating Shutdown Sequence

A socket connection can be taken down in one of several ways. [WSPShutdown](#) (with *how* equal to SD\_SEND or SD\_BOTH), and [WSPSendDisconnect](#) may be used to initiate a graceful connection shutdown.

[WSPCloseSocket](#) can be used to initiate either a graceful or abortive shutdown, depending on the linger options invoked by the closing a socket. See below for more information about graceful and abortive shutdowns, and linger options.

## Indicating Remote Shutdown

Service providers indicate connection teardown that is initiated by the remote party through the FD\_CLOSE network event. Graceful shutdown is also be indicated through [WSPRecv](#) when the number of bytes read is 0 for byte-stream protocols, or through a return error code of WSAEDISCON for message-oriented protocols. In any case, a [WSPRecv](#) return error code of WSAECONNRESET indicates an abortive shutdown.

## Exchanging User Data at Shutdown Time

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

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

CLIENT SIDE	SERVER SIDE
(1) Invokes <a href="#">WSPSendDisconnect</a> to conclude session and supply transaction total.	
	(2) Gets FD_CLOSE, or <a href="#">WSPRecv</a> with a return value of zero or WSAEDISCON indicating graceful shutdown in progress.
	(3) Invokes <a href="#">WSPRecvDisconnect</a> to get client's transaction total.
	(4) Computes cumulative grand total of all transactions.
	(5) Invokes <a href="#">WSPSendDisconnect</a> to transmit grand total.

CLIENT SIDE	SERVER SIDE
(6) Receives FD_CLOSE indication.	(5') Invokes <b>WSPCloseSocket</b>
(7) Invokes <b>WSPRecvDisconnect</b> to receive and store cumulative grand total of transactions.	
	(8) Invokes <b>WSPCloseSocket</b>

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

# Graceful Shutdown, Linger Options, and Socket Closure in the SPI

3/5/2021 • 3 minutes to read • [Edit Online](#)

It is important to distinguish between shutting down a socket connection and closing a socket. Shutting down a socket connection involves an exchange of protocol messages between the two endpoints, which is hereafter referred to as a shutdown sequence. Two general classes of shutdown sequences are defined: graceful and abortive. 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 [WSPShutdown](#) function and the [WSPSendDisconnect](#) function can be used to initiate a shutdown sequence, while the [WSPCloseSocket](#) function is used to deallocate socket handles and free up any associated resources. Some amount of confusion arises, however, from the fact that the [WSPCloseSocket](#) 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 [WSPCloseSocket](#) to both initiate the shutdown sequence and deallocate the socket handle.

To facilitate this usage, the sockets interface provides for controls through the socket option mechanism that allows the programmer to indicate whether the implicit shutdown sequence should be graceful or abortive, and also whether the function should linger (that is, not complete immediately) to allow time for a graceful shutdown sequence to complete.

By establishing appropriate values for the socket options SO\_LINGER and SO\_DONTLINGER, the following types of behavior can be obtained with the [WSPCloseSocket](#) function.

- Abortive shutdown sequence, immediate return from [WSPCloseSocket](#).
- Graceful shutdown, delay return until either shutdown sequence completes or a specified time interval elapses. If the time interval expires before the graceful shutdown sequence completes, an abortive shutdown sequence occurs and [WSPCloseSocket](#) returns.
- Graceful shutdown, return immediately, and allow the shutdown sequence to complete in the background. This is the default behavior. Note, however, that the application has no way of knowing when (or whether) the graceful shutdown sequence completes.

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

CLIENT SIDE	SERVER SIDE
(1) Invokes <a href="#">WSPShutdown</a> ( <i>s</i> , SD_SEND) to signal end of session and that client has no more data to send.	
	(2) Receives FD_CLOSE, indicating graceful shutdown in progress and that all data has been received.

CLIENT SIDE	SERVER SIDE
	(3) Sends any remaining response data.
(5') Gets FD_READ and invoke recv to get any response data sent by server.	(4) Invokes <a href="#">WSPShutdown</a> ( <i>s</i> , SD_SEND) to indicate server has no more data to send.
(5) Receives FD_CLOSE indication.	(4') Invokes <a href="#">WSPCloseSocket</a>
(6) Invokes <a href="#">WSPCloseSocket</a>	

The timing sequence is maintained from step (1) to step (6) between the client and the server, except for steps (4') and (5') which only have 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.

# Socket Connections on Connectionless Protocols

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following sections describe the semantics of using connect operations on connectionless protocols such as UDP and IPX.

- [Connecting to a Default Peer](#)
- [Reconnecting and Disconnecting](#)
- [Using Sendto While Connected](#)

# Connecting to a Default Peer

3/5/2021 • 2 minutes to read • [Edit Online](#)

For a socket bound to a connectionless protocol, the operation performed by [WSPConnect](#) is merely to establish a default destination address so that the socket may be used with subsequent connection-oriented send and receive operations ([WSPSend](#) and [WSPRecv](#)). Any datagrams received from an address other than the destination address specified will be discarded.

# Reconnecting and Disconnecting

3/5/2021 • 2 minutes to read • [Edit Online](#)

The default destination may be changed by simply calling [WSPConnect](#) again, even if the socket is already connected. Any datagrams queued for receipt are discarded if the new address is different from the address specified in a previous [WSPConnect](#).

If the address supplied for the socket is all zeros, the socket will be disconnected — the default remote address will be indeterminate, so [WSPSend](#) and [WSPRecv](#) calls will return the error code WSAENOTCONN, although [WSPSendTo](#) and [WSPRecvFrom](#) may still be used.

# Using Sendto While Connected

3/5/2021 • 2 minutes to read • [Edit Online](#)

**WSPSendTo** will always deliver the data to the specified address, even though a designated peer for the sending socket has been established in **WSPConnect**.

# Socket I/O

3/5/2021 • 2 minutes to read • [Edit Online](#)

There are three primary ways of doing I/O in Windows Sockets 2:

- Blocking I/O.
- Nonblocking I/O along with asynchronous notification of network events.
- Overlapped I/O with completion indication.

We describe each method in the following sections. Blocking I/O is the default behavior, nonblocking mode can be used on any socket that is placed into nonblocking mode, and overlapped I/O can only occur on sockets that are created with the overlapped attribute. It is also interesting to note that the two calls for sending: [WSPSend](#) and [WSPSendTo](#) and the two calls for receiving: [WSPRecv](#) and [WSPRecvFrom](#) each implement all three methods of I/O. Service providers determine how to perform the I/O operation based on socket modes, attributes, and the input parameter values.

# Blocking Input/Output

3/5/2021 • 2 minutes to read • [Edit Online](#)

The simplest form of I/O in Windows Sockets 2 is blocking I/O. As mentioned in [Socket Attribute Flags and Modes](#), sockets are created in blocking mode by default. Any I/O operation with a blocking socket will not return until the operation has been fully completed. Thus, any thread can only execute one I/O operation at a time. For example, if a thread issues a receive operation and no data is currently available, the thread will block until data becomes available and is placed into the thread's buffer. Although this is simple, it is not necessarily the most efficient way to do I/O (see [Pseudo Blocking and True Blocking](#) for more information).

# Nonblocking Input/Output

3/5/2021 • 2 minutes to read • [Edit Online](#)

If a socket is in nonblocking mode, any I/O operation must either complete immediately or return the error code WSAEWOULDBLOCK indicating that the operation cannot be finished right away. In the latter case, a mechanism is needed to discover when it is appropriate to try the operation again with the expectation that the operation will succeed. A set of network events has been defined for this purpose. These events can be polled or waited on by using [WSPSelect](#), or they can be registered for asynchronous delivery by calling [WSPAyncSelect](#) or [WSPEventSelect](#). See [Notification of Network Events](#) for more information.

# Overlapped Input/Output

3/5/2021 • 6 minutes to read • [Edit Online](#)

Windows Sockets 2 introduces overlapped I/O and requires that all transport providers support this capability. Overlapped I/O can be performed only on sockets that were created through the [WSocket](#) function with the WSA\_FLAG\_OVERLAPPED flag set, and follow the model established in Windows.

For receiving, a client uses [WSPRecv](#) or [WSPRecvFrom](#) to supply buffers into which data is to be received. If one or more buffers are posted prior to the time when data has been received by the network, it is possible that data will be placed into the user's buffers immediately as it arrives and thereby avoid the copy operation that would otherwise occur. If data arrives when receive buffers have already been posted, it is copied immediately into the user's buffers. If data arrives when no receive buffers have been posted by the application, the service provider resorts to the synchronous style of operation where the incoming data is buffered internally until such time as the client issues a receive call and thereby supplies a buffer into which the data may be copied. An exception to this would be if the application used [WSPSetSockOpt](#) 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, clients use [WSPSend](#) or [WSPSendTo](#) to supply pointers to filled buffers and then agree not to 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 completed immediately and that the corresponding completion indication has already occurred. That is, the associated event object has been signaled, or the completion routine has been queued through [WPUQueueApc](#). 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 receive buffers are filled. Any other error code indicates that the overlapped operation was not successfully initiated and that no completion indication will be forthcoming.

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

The deferred completion feature of overlapped I/O is also available for [WSPIoctl](#).

## Delivering Completion Indications

Service providers have two ways to indicate overlapped completion: setting a client-specified event object, or invoking a client-specified completion routine. In both cases a data structure, [WSAOVERLAPPED](#), is associated with each overlapped operation. This structure is allocated by the client and used by it to indicate which event object (if any) is to be set when completion occurs. The [WSAOVERLAPPED](#) structure may be used by the service provider as a place to store a handle to the results (for example, number of bytes transferred, updated flags, error codes, etc.) for a particular overlapped operation. To obtain these results clients must invoke [WSPGetOverlappedResult](#), passing in a pointer to the corresponding overlapped structure.

If event based completion indication is selected for a particular overlapped I/O request, the [WSPGetOverlappedResult](#) routine may itself be used by clients to either poll or wait for completion of the overlapped operation. If completion-routine-based completion indication is selected for a particular overlapped

I/O request, only the polling option of **WSPGetOverlappedResult** is available. A client may also use other means to wait (such as using **WSAWaitForMultipleEvents**) until the corresponding event object has been signaled or the specified completion routine has been invoked by the service provider. Once completion has been indicated, the client may invoke **WSPGetOverlappedResult**, with the expectation that the call will complete immediately.

## Invoking Socket I/O Completion Routines

If the *lpCompletionRoutine* parameter to an overlapped operation is not **NULL**, it is the service provider's responsibility to arrange for invocation of the client-specified completion routine when the overlapped operation completes. Since the completion routine must be executed in the context of the same thread that initiated the overlapped operation, it cannot be invoked directly from the service provider. The Ws2\_32.DLL offers an asynchronous procedure call (APC) mechanism to facilitate invocation of completion routines.

A service provider arranges for a function to be executed in the proper thread by calling **WPUQueueApc**. This function can be called from any process and thread context, even a context different from the thread and process that was used to initiate the overlapped operation.

**WPUQueueApc** takes as input parameters a pointer to a **WSATHREADID** structure, a pointer to an APC function to be invoked, and a 32-bit context value that is subsequently passed to the APC function. Service providers are always supplied with a pointer to the proper **WSATHREADID** structure through the *lpThreadId* parameter to the overlapped function. The provider should store the **WSATHREADID** structure locally and supply a pointer to this copy of the **WSATHREADID** structure as an input parameter to **WPUQueueApc**. Once the **WPUQueueApc** function returns, the provider can dispose of its copy of the **WSATHREADID**.

The procedure **WPUQueueApc** simply enqueues sufficient information to call the indicated APC function with the given parameters, but does not call it. When the target thread enters an alertable wait state, this information is dequeued and a call is made to the APC function in that target thread and process context. Because the APC mechanism supports only a single 32-bit context value, the APC function cannot itself be the client-specified completion routine, which involves more parameters. The service provider must instead supply a pointer to its own APC function which uses the supplied context value to access the needed result information for the overlapped operation, and then invokes the client-specified completion routine.

For service providers where a user-mode component implements overlapped I/O, a typical usage of the APC mechanism is as follows:

- When the I/O operation completes, the provider allocates a small buffer and packs it with a pointer to the client-supplied completion procedure and parameter values to pass to the procedure.
- It queues an APC, specifying the pointer to the buffer as the context value and its own intermediate procedure as the target procedure.
- When the target thread eventually enters alertable wait state, the service provider's intermediate procedure is called in the proper thread context.
- The intermediate procedure simply unpacks parameters, deallocates the buffer, and calls the client-supplied completion procedure.
- For service providers where a kernel-mode component implements overlapped I/O, a typical implementation is similar, except that the implementation would use standard kernel interfaces to enqueue the APC.

Description of the relevant kernel interfaces is outside the scope of the Windows Sockets 2 specification.

### NOTE

Service providers must allow Windows Sockets 2 clients 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.

Under some circumstances, a layered service provider may need to initiate and complete overlapped operations from within an internal worker thread. In this case, a [WSATHREADID](#) would not be available from an incoming function call. The service provider interface provides an upcall, [WPUOpenCurrentThread](#), to obtain a [WSATHREADID](#) for the current thread. When this [WSATHREADID](#) is no longer needed, its resources should be returned by calling [WPUCloseThread](#).

# Summary of Overlapped Completion Indication Mechanisms in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following table summarizes the completion semantics for an overlapped socket, showing the various combination of *lpOverlapped*, **hEvent**, and *lpCompletionRoutine*.

<i>LPOVERLAPPED</i>	<b>HEvent</b>	<i>LPCOMPLETIONROUTINE</i>	<b>COMPLETION INDICATION</b>
NULL	Not applicable	Ignored	Operation completes synchronously, that is, it behaves as if it were a nonoverlapped socket.
not NULL	NULL	NULL	Operation completes overlapped, but there is no Windows Sockets 2-supported completion mechanism. The completion port mechanism (if supported) may be used in this case, otherwise there will be no completion notification.
not NULL	not NULL	NULL	Operation completes overlapped, notification by signaling event object.
not NULL	Ignored	not NULL	Operation completes overlapped, notification by scheduling completion routine.

# Support for Scatter/Gather Input/Output in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

The [WSPSend](#), [WSPSendTo](#), [WSPRecv](#), and [WSPRecvFrom](#) routines all take an array of client buffers as input parameters and thus may be used for scatter/gather (or vectored) I/O. This can be very useful in instances where portions of each message being transmitted consist of one or more fixed length header components in addition to a message body. Such header components need not be concatenated 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.

Utilizing lists of buffers instead of a single buffer does not change the boundaries that apply to receive operations. For message-oriented protocols, a receive operation completes whenever a single message has been received, regardless of how many or few of the supplied buffers were used. Likewise for stream-oriented protocols, a receive completes when an unspecified quantity of bytes arrives over the network, not necessarily when all of the supplied buffers are full.

# Out-of-Band Data in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

The service providers which support the out-of-band data (OOB) abstraction for the stream-style sockets must adhere to the semantics in this section. We will describe OOB data handling in a protocol-independent manner. Please refer to the [Winsock Annexes](#) for a discussion of OOB data implemented using urgent data in TCP/IP service providers. In the following, the use of [WSRecv](#) also applies to [WSRecvFrom](#).

# Protocol Independent Out-of-Band Data in the SPI

3/5/2021 • 3 minutes to read • [Edit Online](#)

Out-of-Band (OOB) data is a logically independent transmission channel associated with a pair of connected stream sockets. OOB data may be delivered to the user independently of normal data. The abstraction defines that the OOB data facilities must support the reliable delivery of at least one OOB data block at a time. This data block may contain at least one byte of data, and at least one OOB data block may be pending delivery to the user at any one time. For communications protocols which support in-band signaling (that is, 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 OOB data.

A user can determine if there is any OOB data waiting to be read using the [WSPIoctl](#) (SIOCATMARK) function. For protocols where the concept of the position of the OOB data block within the normal data stream is meaningful (that is, 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 [WSPIoctl](#) (SIOCATMARK) functionality — the presence or absence of OOB data is all that is required.

For protocols where the concept of the position of the OOB data block within the normal data stream is meaningful an application may prefer to process out-of-band data inline, as part of the normal data stream. This is achieved by setting the socket option SO\_OOBINLINE (see section [WSPIoctl](#)). 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 WSPIoctl command to determine whether there is any unread OOB data preceding the mark. For example, it might use this to resynchronize with its peer by ensuring that all data up to the mark in the data stream is discarded when appropriate.

With SO\_OOBINLINE disabled (by default):

- The Winsock service provider notifies a client of an FD\_OOB event, if the client registered for notification with [WSPAsyncSelect](#), in exactly the same way FD\_READ is used to notify of the presence of normal data. That is, FD\_OOB is posted when OOB data arrives and there was no OOB data previously queued, and also when data is read using the MSG\_OOB flag, and some OOB data remains to be read after the read operation has returned. FD\_READ messages are not posted for OOB data.
- The Winsock service provider returns from [WSPSelect](#) with the appropriate *exceptfds* socket set if OOB data is queued on the socket.
- The client can call [WSPRecv](#) with MSG\_OOB to read the urgent data block at any time. The block of OOB data jumps the queue.
- The client can call [WSPRecv](#) 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 [WSPRecv](#), the service provider notifies the client with FD\_OOB or through *exceptfds* when using [WSPSelect](#).
- For protocols where the OOB data has a position within the normal data stream, a single [WSPRecv](#) operation will not span that position. One [WSPRecv](#) will return the normal data before the mark, and a second [WSPRecv](#) is required to begin reading data after the mark.

With SO\_OOBINLINE enabled:

- FD\_OOB messages are not posted for OOB data — for the purpose of the [WSPSelect](#) and [WSPAsyncSelect](#) functions, OOB data is treated as normal data, and indicated by setting the socket in *readfds* or by sending an

FD\_READ message respectively.

- The client may not call [WSRecv](#) with the MSG\_OOB flag set to read the OOB data block — the error code WSAEINVAL will be returned.
- The client can call [WSRecv](#) 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 OOB data when SO\_OOBINLINE is off.

# Shared Sockets in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

Socket sharing between processes in Windows Sockets is implemented as follows. A source process calls [WSPDuplicateSocket](#) to obtain a special [WSAPROTOCOL\\_INFO](#) structure. It uses some interprocess communications (IPC) mechanism to pass the contents of this structure to a target process. The target process then uses the [WSAPROTOCOL\\_INFO](#) structure in a call to [WSPSocket](#). The socket descriptor returned by this function will be an additional socket descriptor to an underlying socket which thus becomes shared.

It is the service provider's responsibility to perform whatever operations are needed in the source process context and to create a [WSAPROTOCOL\\_INFO](#) structure that will be recognized when it subsequently appears as a parameter to [WSPSocket](#) in the target processes' context. The **dwProviderReserved** member of the [WSAPROTOCOL\\_INFO](#) structure is available for the service provider's use, and may be used to store any useful context information, including a duplicated handle.

This mechanism is designed to be appropriate for both single-threaded and preemptive multithreaded versions of Windows. Note however, that sockets may be shared among threads in a given process without using the [WSPDuplicateSocket](#) function, since a socket descriptor is valid in all of a process' threads.

As is described in section [Descriptor Allocation](#), when new socket descriptors are allocated IFS providers must call [WPUModifyIFSHandle](#) and non-IFS providers must call [WPUCreateSocketHandle](#).

One possible scenario for establishing and using a shared socket in a handoff mode is illustrated in the following table.

SOURCE PROCESS	IPC	DESTINATION PROCESS
1) <a href="#">WSPSocket</a> , <a href="#">WSPConnect</a>		
2) Requests target process identifier.	=>	
		3) Receives process identifier request and responds.
4) Receives process identifier.	<=	
5) Calls <a href="#">WSPDuplicateSocket</a> to get a special <a href="#">WSAPROTOCOL_INFO</a> structure.		
6) Sends <a href="#">WSAPROTOCOL_INFO</a> structure to target.		
	=>	7) Receives <a href="#">WSAPROTOCOL_INFO</a> structure.
		8) Calls <a href="#">WSPSocket</a> to create shared socket descriptor.
		9) Uses shared socket for data exchange.

SOURCE PROCESS	IPC	DESTINATION PROCESS
10) <a href="#">WSPClosesocket</a>	<==	

# Multiple Handles to a Single Socket

3/5/2021 • 2 minutes to read • [Edit Online](#)

Since what are duplicated are the socket descriptors and not the underlying sockets, all of the states associated with a socket are held in common across all the descriptors. For example a [WSPSetSockOpt](#) operation performed using one descriptor is subsequently visible using a [WSPPGetSockOpt](#) from any or all descriptors.

Notification on shared sockets is subject to the usual constraints of [WSPAsyncSelect](#) and [WSPEventSelect](#). 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 consider using threads instead of separate processes.

# Reference Counting

3/5/2021 • 2 minutes to read • [Edit Online](#)

A process may call **WSPCloseSocket** on a duplicated socket and the descriptor will become deallocated. The underlying socket, however, will remain open until **WSPCloseSocket** is called on the last remaining descriptor.

# Precedence Guidelines

3/5/2021 • 2 minutes to read • [Edit Online](#)

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

The general guideline for supporting multiple outstanding operations on shared sockets is that a service provider is encouraged to honor all simultaneous operations on shared sockets, especially the most recent operation on a socket object. If need be, this may occur at the expense of canceling some of the previous but still pending operations, which will return WSAEINTR in this case.

# Protocol-Independent Multicast and Multipoint in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

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

Current multipoint implementations (for example, IP multicast, ST-II, T.120, ATM UNI) vary widely with respect to how nodes join a multipoint session, whether a particular node is designated as a central or root node, and whether data is exchanged between all nodes or only between a root node and various leaf nodes. The Windows Sockets 2 [WSAPROTOCOL\\_INFO](#) structure is used to declare the multipoint attributes of a protocol. By examining these attributes the programmer will know what conventions to follow in using the applicable Winsock functions to set up, use, and tear down multipoint sessions.

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

- Three attribute bits in the [WSAPROTOCOL\\_INFO](#) structure.
- Four flags defined for the *dwFlags* parameter of [WSPSocket](#)
- One function, [WSPJoinLeaf](#), for adding leaf nodes into a multipoint session.
- Two [WSPIoctl](#) 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 a service provider from also supporting an existing protocol-dependent interface, such as the Deering socket options for IP multicast.

# Multipoint Taxonomy and Glossary

3/5/2021 • 3 minutes to read • [Edit Online](#)

The taxonomy described below 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 among session participants.

In the control plane, there are two distinct types of session establishment:

- rooted
- nonrooted

For a rooted control plane, 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 nonrooted control plane, all members belonging to a multipoint session are leaves, that is, 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 OOB mechanism which is outside the scope of this discussion (and hence not addressed in the proposed Windows Sockets extensions). 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 nonrooted control plane could also be considered to be implicitly rooted. Examples of this kind of implicitly rooted multipoint schemes are: a teleconferencing bridge, the IP multicast system, a Multipoint Control Unit (MCU) in an H.320 video conference, etc.

In the data plane, there are also two types of data transfer styles:

- rooted
- nonrooted

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 bidirectional. 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 uses a rooted data plane is ST-II.

In a nonrooted 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 an example of a protocol that uses a nonrooted 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. As such, they are irrelevant to the interface the client would use to perform multipoint communications. Therefore these issues are not addressed by the Windows Sockets interface.

The following table depicts the taxonomy described above and indicates how existing schemes fit into each of

the control and data plane categories. Note that there does not appear to be any multipoint schemes that employ a nonrooted control plane along with a rooted data plane.

DATA PLANE	ROOTED CONTROL PLANE EXAMPLES	NONROOTED (IMPLICIT ROOTED) CONTROL PLANE EXAMPLES
Rooted data plane	ATM, ST-II	No known examples
Nonrooted data plane	T.120	IP-multicast, H.320 (MCU)

# Multipoint attributes in WSAPROTOCOL\_INFO

3/5/2021 • 2 minutes to read • [Edit Online](#)

Three attribute parameters are defined in the [WSAPROTOCOL\\_INFO](#) structure in order to distinguish the different schemes used in the control and data planes, respectively:

- XP1\_SUPPORT\_MULTIPOINT with a value of 1 indicates this protocol entry supports multipoint communications, and that the following two parameters are meaningful.
- XP1\_MULTIPOINT\_CONTROL\_PLANE indicates whether the control plane is rooted (value = 1) or nonrooted (value = 0).
- XP1\_MULTIPOINT\_DATA\_PLANE indicates whether the data plane is rooted (value = 1) or nonrooted (value = 0).

Two [WSAPROTOCOL\\_INFO](#) entries would be present if a multipoint protocol supported both rooted and nonrooted data planes, one entry for each.

# Multipoint Socket Attributes

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 client to be able to indicate the intended use of a socket coincident with its creation. This is done through four multipoint attribute flags that can be set through the *dwFlags* parameter in [WSPSocket](#):

- 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.

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 [WSPSocket](#)'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 ioctl

3/5/2021 • 2 minutes to read • [Edit Online](#)

When d\_leaf sockets are used in a nonrooted 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 [WSPIoctl](#) is used to enable or disable loopback of multipoint traffic.

# SIO\_MULTICAST\_SCOPE ioctl

3/5/2021 • 2 minutes to read • [Edit Online](#)

When multicasting is employed, it is usually necessary to specify the scope over which the multicast should occur. Here scope is defined to be the number of routed network segments to be covered. The SIO\_MULTICAST\_SCOPE command code for [WSPIoctl](#) is used to control this. 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.

# SPI Semantics for Joining Multipoint Leaves

3/5/2021 • 2 minutes to read • [Edit Online](#)

In the following, a multipoint socket is frequently characterized by describing 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 a reference to a c\_root socket could refer to 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 **WSPJoinLeaf** 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 **WSPListen** 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 **WSPAccept**. 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 client 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 **WSPJoinLeaf** to initiate the connection and request admittance. The root client receives FD\_ACCEPT when an incoming admittance request arrives, and admits the leaf node by calling **WSPAccept**. The leaf node receives FD\_CONNECT when it has been admitted.

In a nonrooted control plane, where all nodes are c\_leafs, the **WSPJoinLeaf** function 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 usable in the multipoint session. In the case of IP multicast, this would correspond to the IP\_ADD\_MEMBERSHIP socket option.

There are, therefore, three instances where a client would use **WSPJoinLeaf**:

- Acting as a multipoint root and inviting a new leaf to join the session.
- Acting as a leaf making an admittance request to a rooted multipoint session.
- Acting as a leaf seeking admittance to a nonrooted multipoint session (for example, IP multicast.)

# Using WSPJoinLeaf

3/5/2021 • 2 minutes to read • [Edit Online](#)

As mentioned previously, **WSPJoinLeaf** is used to join a leaf node into a multipoint session. **WSPJoinLeaf** has the same parameters and semantics as **WSPConnect** except that it returns a socket descriptor (as in **WSPAccept**), 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 nonblocking mode, the returned socket descriptor will not be usable until after a corresponding FD\_CONNECT indication has been received. A root application in a multipoint session may call **WSPJoinLeaf** 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 **WSPJoinLeaf** 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 network event indications (for example FD\_CLOSE) for the connection that exists to the particular *c\_leaf*. Some multipoint implementations may also allow this socket to be used for side chats between the root and an individual leaf node. An FD\_CLOSE indication should be given for this socket if the corresponding leaf node calls **WSPCloseSocket** to drop out of the multipoint session. Symmetrically, invoking **WSPCloseSocket** on the *c\_leaf* socket returned from **WSPJoinLeaf** will cause the socket in the corresponding leaf node to get FD\_CLOSE notification.

When **WSPJoinLeaf** 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 a rooted control scheme, the root client would put its *c\_root* socket in the listening mode by calling **WSPListen**. The standard FD\_ACCEPT notification will be delivered when the leaf node requests to join itself to the multipoint session. The root client uses the usual **WSPAccept** function to admit the new leaf node. The value returned from **WSPAccept** is also a *c\_leaf* socket descriptor just like those returned from **WSPJoinLeaf**. To accommodate multipoint schemes that allow both root-initiated and leaf-initiated joins, it is acceptable for a *c\_root* socket that is already in listening mode to be used as in input to **WSPJoinLeaf**.

A multipoint root client is generally responsible for the orderly dismantling of a multipoint session. Such an application may use **WSPShutdown** or **WSPCloseSocket** on a *c\_root* socket to cause all of the associated *c\_leaf* sockets, including those returned from **WSPJoinLeaf** 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 SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

In the control plane, there are some significant semantic differences between a c\_root socket and a regular point-to-point socket:

- The c\_root socket can be used in [WSPJoinLeaf](#) to join a new leaf.
- Placing a c\_root socket into the listening mode (by calling [WSPListen](#)) does not preclude the c\_root socket from being used in a call to [WSPJoinLeaf](#) to add a new leaf, or for sending and receiving multipoint data.
- The closing of a c\_root socket will cause all the associated c\_leaf sockets to get FD\_CLOSE notification.

There are 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 [WSPJoinLeaf](#), and the use of c\_leaf socket in [WSPListen](#) 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

- The data sent on the d\_root socket will be delivered to all the leaves in the same multipoint session.
- 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 nonrooted data plane, the data sent on the d\_leaf socket will go to all of the other leaf nodes, and the data received could be from any of the other leaf nodes. As mentioned earlier, the information about whether the d\_leaf socket is in a rooted or nonrooted data plane is contained in the corresponding [WSAPROTOCOL\\_INFO](#) structure for the socket.

# Socket Options and IOCTLs

3/5/2021 • 2 minutes to read • [Edit Online](#)

Some of the socket options for Windows Sockets 2 are summarized in the following table. More detailed information is provided in section 4 under [WSPGetSockOpt](#) and/or [WSPSetSockOpt](#). There are other new protocol-specific socket options which can be found in the Protocol-Specific Annex. A complete list of [Socket Options](#) for Windows Sockets are available in the Winsock reference.

For a summary of some of the Winsock ioctls, see [Summary of Socket ioctl Opcodes](#). A complete list of [Winsock IOCTLs](#) are available in the Winsock reference.

## Summary of Common Socket Options

A Winsock service provider must recognize all of these options, and (for [WSPGetSockOpt](#)) return plausible values for each. The default value for each option is shown in the following table.

Value

Type

Meaning

Default

Note

SO\_ACCEPTCONN

BOOL

Socket is listening.

FALSE unless a [WSPListen](#) has been performed.

SO\_BROADCAST

BOOL

Socket is configured for the transmission and receipt of broadcast messages.

FALSE

SO\_DEBUG

BOOL

Debugging is enabled.

FALSE

(i)

SO\_DONTLINGER

BOOL

If true, the SO\_LINGER option is disabled.

TRUE

**SO\_DONTROUTE**

BOOL

Routing is disabled. Succeeds but is ignored on AF\_INET sockets; fails on AF\_INET6 sockets with [WSAENOPROTOOPT](#). Not supported on ATM sockets (results in an error).

FALSE

(i)

**SO\_ERROR**

int

Retrieves error status and clear.

0

**SO\_GROUP\_ID**

GROUP

Reserved.

NULL

Get only

**SO\_GROUP\_PRIORITY**

int

Reserved.

0

**SO\_KEEPALIVE**

BOOL

Keepalives are being sent. Not supported on ATM sockets (results in an error).

FALSE

(i)

**SO\_LINGER**

Structure linger

Returns the current linger options.

l\_onoff is 0

**SO\_MAX\_MSG\_SIZE**

int

Maximum outbound size of a message for message socket types. There is no provision to determine the maximum inbound message size. Has no meaning for stream-oriented sockets.

Implementation dependent

Get only

SO\_OOBINLINE

BOOL

OOB data is being received in the normal data stream.

FALSE

SO\_PROTOCOL\_INFOW

structure [WSAPROTOCOL\\_INFO](#)

Description of protocol information for the protocol that is bound to this socket.

Protocol dependent

Get only

SO\_RCVBUF

int

The total per-socket buffer space reserved for receives. This is unrelated to SO\_MAX\_MSG\_SIZE and does not necessarily correspond to the size of the TCP receive window.

Implementation dependent

(i)

SO\_REUSEADDR

BOOL

The address to which this socket is bound can be used by others. Not applicable on ATM sockets.

FALSE

SO\_SNDBUF

int

The total per-socket buffer space reserved for sends. This is unrelated to SO\_MAX\_MSG\_SIZE and does not necessarily correspond to the size of a TCP send window.

Implementation dependent

(i)

SO\_TYPE

int

The type of the socket (for example, SOCK\_STREAM).

As created through socket.

PVD\_CONFIG

char FAR \*

An opaque data structure object containing configuration information of the service provider.

Implementation dependent

TCP\_NODELAY

BOOL

Disables the Nagle algorithm for send coalescing.

Implementation dependent

(i) A service provider may silently ignore this option on [WSPSetSockOpt](#) and return a constant value for [WSPGetSockOpt](#), or it may accept a value for [WSPSetSockOpt](#) and return the corresponding value in [WSPGetSockOpt](#) without using the value in any way.

## Related topics

[Socket Options](#)

[SOL\\_SOCKET Socket Options](#)

[IPPROTO\\_TCP Socket Options](#)

[IPPROTO\\_UDP Socket Options](#)

[Winsock IOCTLs](#)

# Summary of Socket Ioctl Opcodes

3/5/2021 • 2 minutes to read • [Edit Online](#)

Some of the socket IOCTL opcodes for Windows Sockets 2 are summarized in the following table. More detailed information is in the Winsock reference on [Winsock IOCTLs](#) and the [WSPIoctl](#) function. There are other new protocol-specific IOCTL opcodes that can be found in the protocol-specific annex.

A complete list of [Winsock IOCTLs](#) are available in the Winsock reference.

OPCODE	INPUT TYPE	OUTPUT TYPE	MEANING
FIONBIO	Unsigned long		Enables or disables nonblocking mode on the socket.
FIONREAD		Unsigned long	Determines the amount of data that can be read atomically from the socket.
SIOCATMARK		BOOL	Determines whether or not all OOB data has been read.
SIO_ASSOCIATE_HANDLE	Companion API dependent		Associates the socket with the specified handle of a companion interface.
SIO_ENABLE_CIRCULAR_QUEUEING			Enables circular queuing.
SIO_FIND_ROUTE	<a href="#">sockaddr</a> structure		Requests the route to the specified address to be discovered.
SIO_FLUSH			Discards current contents of the sending queue.
SIO_GET_BROADCAST_ADDRESS		<a href="#">sockaddr</a> structure	Retrieves the protocol-specific broadcast address to be used in <a href="#">WSPSendTo</a> .
SIO_GET_QOS		<a href="#">QOS</a>	Retrieves current flow specifications for the socket.
SIO_GET_GROUP_QOS		<a href="#">QOS</a>	Reserved.
SIO_MULTIPOINT_LOOPBACK	BOOL		Controls whether data sent in a multipoint session will also be received by the same socket on the local host.

OPCODE	INPUT TYPE	OUTPUT TYPE	MEANING
SIO_MULTICAST_SCOPE	int		Specifies the scope over which multicast transmissions will occur.
SIO_SET_QOS	<a href="#">QOS</a>		Establishes new flow specifications for the socket.
SIO_SET_GROUP_QOS	<a href="#">QOS</a>		Reserved.
SIO_TRANSLATE_HANDLE	int	Companion-API dependent	Obtains a corresponding handle for socket <i>s</i> that is valid in the context of a companion interface.
SIO_ROUTING_INTERFACE_QUERY	<a href="#">sockaddr</a>	<a href="#">sockaddr</a>	Obtains the address of the local interface that should be used to send to the specified address.
SIO_ROUTING_INTERFACE_CHANGE	<a href="#">sockaddr</a>		Requests notification of changes in information reported through SIO_ROUTING_INTERFACE_QUERY for the specified address.
<a href="#">SIO_ADDRESS_LIST_QUERY</a>		<a href="#">SOCKET_ADDRESS</a>	Obtains a list of local transport addresses of the socket's protocol family to which the application can bind. The list of addresses varies based on address family and some addresses are excluded from the list.
SIO_ADDRESS_LIST_CHANGE			Requests notification of changes in information reported through SIO_ADDRESS_LIST_QUERY
SIO_QUERY_PNP_TARGET_HANDLE		SOCKET	Obtains socket descriptor of the next provider in the chain on which current socket depends in regards to PnP.

## Related topics

[Winsock IOCTLs](#)

[WSPIoctl](#)

# Using SIO\_ADDRESS\_LIST\_SORT

3/5/2021 • 2 minutes to read • [Edit Online](#)

The **SIO\_ADDRESS\_LIST\_SORT** IOCTL allows application developers to sort a list of IPv6 and IPv4 destination addresses to determine the best available address for making a connection. The **SIO\_ADDRESS\_LIST\_SORT** IOCTL is supported on Windows XP and later.

On Windows Vista and later, the [CreateSortedAddressPairs](#) function takes a supplied list of potential IP destination addresses, pairs the destination addresses with the host machine's local IP addresses, and sorts the pairs according to which address pair is best suited for communication between the two peers. The [CreateSortedAddressPairs](#) function should be used instead of the **SIO\_ADDRESS\_LIST\_SORT** IOCTL on Windows Vista and later.

The following sections describe usage considerations for **SIO\_ADDRESS\_LIST\_SORT**.

## Parameters

The buffer passed to **SIO\_ADDRESS\_LIST\_SORT** is a [SOCKET\\_ADDRESS\\_LIST](#) structure. Each [SOCKET\\_ADDRESS](#) in the list must be in [SOCKADDR\\_IN6](#) format.

The **SIO\_ADDRESS\_LIST\_SORT** IOCTL sorts both IPv6 and IPv4 addresses on Windows Vista and later. Any IPv4 addresses in the list to be sorted must be in the IPv4-mapped IPv6 address format. For more information on the IPv4-mapped IPv6 address format, see [Dual-Stack Sockets](#).

On Windows Server 2003, and Windows XP, **SIO\_ADDRESS\_LIST\_SORT** sorts only IPv6 addresses. IPv4 addresses in the IPv4-mapped IPv6 address format are not supported.

On output, the **iAddressCount** member of the [SOCKET\\_ADDRESS\\_LIST](#) structure may be smaller than on input if the IOCTL code determines that some destination addresses are invalid.

## Sorting Determination

The sorting order for IPv6 addresses for the **SIO\_ADDRESS\_LIST\_SORT** IOCTL is based on the prefix policy table. The prefix policy table is configured using the *Netsh.exe* command line utility. The following command line snippets illustrate basic *Netsh.exe* prefix policy table configuration commands:

```
netsh interface ipv6 show prefixpolicies  
netsh interface ipv6 add prefixpolicy ::/96 3 4  
netsh interface ipv6 delete prefixpolicy ::/96  
netsh interface ipv6 set prefixpolicy ::/96 3 4
```

### NOTE

On Windows Server 2003 and Windows XP, the first netsh command listed above was as follows. All other related commands are the same.

```
netsh interface ipv6 show prefixpolicy
```

Address ordering is also determined by an algorithm outlined in the RFC 3484 on *Default Address Selection for Internet Protocol version 6 (IPv6)* published by the IETF. For more information, see <https://www.ietf.org/rfc/rfc3484.txt>. (This resource may only be available in English.)

The **SIO\_ADDRESS\_LIST\_SORT** IOCTL sorts addresses from best to worst, and fills in **sin6\_scope\_id** members, if needed. For site-local addresses, **SIO\_ADDRESS\_LIST\_SORT** either fills in the scope-id, or removes the address.

The **SIO\_ADDRESS\_LIST\_SORT** IOCTL ignores the source address bound to the socket and only sorts by the destination address list passed as a parameter.

The [CreateSortedAddressPairs](#) function should be used instead of the **SIO\_ADDRESS\_LIST\_SORT** IOCTL on Windows Vista and later. The [CreateSortedAddressPairs](#) function returns a list of address pairs that contains a local source address and a destination address. This provides an application the correct source address to use for each destination address. An application can also filter the results by looking for a specific source address. in the results.

## Requirements

The **SIO\_ADDRESS\_LIST\_SORT** IOCTL is defined in the *Winsock2.h* header file. On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and **SIO\_ADDRESS\_LIST\_SORT** IOCTL is defined in the *Ws2def.h* header file. Note that the *Ws2def.h* header file is automatically included in *Winsock2.h*, and should never be used directly.

The **SIO\_ADDRESS\_LIST\_SORT** IOCTL is supported on Windows XP and later.

## Related topics

[CreateSortedAddressPairs](#)

# Summary of SPI Functions

3/5/2021 • 2 minutes to read • [Edit Online](#)

The SPI functions for Windows Sockets 2 are summarized in the following tables.

- [Generic Data Transport Functions](#)
- [Upcalls Exposed by Windows Sockets 2 DLL](#)
- [Installation and Configuration Functions](#)

# Generic Data Transport Functions

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section lists the data transport functions exposed by Ws2spi.h.

FUNCTION	DESCRIPTION
<a href="#">WSPAccept</a>	An incoming connection is acknowledged and associated with an immediately created socket. The original socket is returned to the listening state. This function also allows for conditional acceptance.
<a href="#">WSPAsyncSelect</a>	Performs asynchronous version of <a href="#">WSPSelect</a> .
<a href="#">WSPBind</a>	Assigns a local name to an unnamed socket.
<a href="#">WSPCancelBlockingCall</a>	Cancels an outstanding blocking Windows Sockets call.
<a href="#">WSPCleanup</a>	Signs off from the underlying Windows Sockets service provider.
<a href="#">WSPCloseSocket</a>	Removes a socket from the per-process object reference table. Only blocks if SO_LINGER is set with a nonzero timeout on a blocking socket.
<a href="#">WSPConnect</a>	Initiates a connection on the specified socket. This function also allows for exchange of connect data and QoS specification.
<a href="#">WSPDuplicateSocket</a>	Returns a <a href="#">WSAPROTOCOL_INFO</a> structure that can be used to create a new socket descriptor for a shared socket.
<a href="#">WSPEnumNetworkEvents</a>	Discovers occurrences of network events.
<a href="#">WSPEventSelect</a>	Associates network events with an event object.
<a href="#">WSPGetOverlappedResult</a>	Gets completion status of overlapped operation.
<a href="#">WSPGetPeerName</a>	Retrieves the name of the peer connected to the specified socket.
<a href="#">WSPGetSockName</a>	Retrieves the local address to which the specified socket is bound.
<a href="#">WSPGetSockOpt</a>	Retrieves options associated with the specified socket.
<a href="#">WSPGetQOSByName</a>	Supplies QoS parameters based on a well-known service name.
<a href="#">WSPIoctl</a>	Provides control for sockets.

FUNCTION	DESCRIPTION
<a href="#">WSPJoinLeaf</a>	Joins a leaf node into a multipoint session.
<a href="#">WSPListen</a>	Listens for incoming connections on a specified socket.
<a href="#">WSPRecv</a>	Receives data from a connected or unconnected socket. This function accommodates scatter/gather I/O, overlapped sockets, and provides the flags parameter as IN/OUT.
<a href="#">WSPRecvDisconnect</a>	Terminates reception on a socket, and retrieve the disconnect data if the socket is connection-oriented.
<a href="#">WSPRecvFrom</a>	Receives data from either a connected or unconnected socket. This function accommodates scatter/gather I/O, overlapped sockets and provides the flags parameter as IN/OUT.
<a href="#">WSPSelect</a>	Performs synchronous I/O multiplexing.
<a href="#">WSPSend</a>	Sends data to a connected socket. This function also accommodates scatter/gather I/O and overlapped sockets.
<a href="#">WSPSendDisconnect</a>	Initiates termination of a socket connection and optionally send disconnect data.
<a href="#">WSPSendTo</a>	Sends data to either a connected or unconnected socket. This function also accommodates scatter/gather I/O and overlapped sockets.
<a href="#">WSPSetSockOpt</a>	Stores options associated with the specified socket.
<a href="#">WSPShutdown</a>	Shuts down part of a full-duplex connection.
<a href="#">WSPSocket</a>	A socket creation function which takes a <a href="#">WSAPROTOCOL_INFO</a> structure as input and allows overlapped sockets to be created.
<a href="#">WSPStartup</a>	Initializes the underlying Windows Sockets service provider.

# Upcalls Exposed by Windows Sockets 2 DLL

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section lists the upcalls that service providers may make into the Windows Sockets client. Service providers receive an upcall dispatch table as a parameter to [WSPStartup](#) function, and use entries in this table to make the upcalls. Therefore, a client does not need to export its WPU functions.

It is not mandatory that providers use all of these upcalls. The following table indicates which upcalls must be used and which are optional.

FUNCTION	DESCRIPTION	STATUS	MEANING
<a href="#">WPUCloseEvent</a>	Closes an open event object handle.	Optional.	The provider may use an appropriate Windows call instead.
<a href="#">WPUCloseSocketHandle</a>	Closes a socket handle allocated by the Windows Sockets DLL.	Required.	The Ws2_32.dll needs to query and/or modify internal state information associated with the socket handle.
<a href="#">WPUCloseThread</a>	Closes a thread ID for an internal service thread.		
<a href="#">WPUCCompleteOverlappedRequest</a>	Delivers overlapped I/O completion notification where the completion mechanism is something other than user mode APC.		
<a href="#">WPUCreateEvent</a>	Creates a new event object.	Optional.	The provider may use an appropriate Windows call instead.
<a href="#">WPUCreateSocketHandle</a>	Creates a new socket handle for nonIFS providers.	Required for nonIFS providers.	The Ws2_32.dll needs to query and/or modify internal state information associated with the socket handle.
<a href="#">WPUFDIsSet</a>	Checks the membership of the specified socket handle.	Optional.	This is just a convenience function that knows how to dig through <b>fd_set</b> structures. A provider may need to dig through these structures explicitly anyway.
<a href="#">WPUGetProviderPath</a>	Retrieves the DLL path for the specified provider.	Required.	Only the Ws2_32.dll would know where an adjacent protocol layer (potentially from another vendor) has been installed.

FUNCTION	DESCRIPTION	STATUS	MEANING
<a href="#">WPUModifyIFSHandle</a>	Receives a (possibly) modified IFS handle from the Windows Sockets DLL.	Required for IFS providers.	The Ws2_32.dll needs to query and/or modify internal state information associated with the socket handle.
<a href="#">WPUPostMessage</a>	Performs the standard <a href="#">PostMessage</a> function in a way that maintains backward compatibility.	Required.	Windows 2000 and Windows NT only. Windows 95 allows post message from kernel mode.
<a href="#">WPUQueryBlockingCalliback</a>	Returns a pointer to a thread's blocking hook function.	Required.	There is no corresponding Windows functionality. Only the Ws2_32.dll has the information to accomplish this.
<a href="#">WPUQuerySocketHandleContext</a>	Gets a socket's context value (nonIFS providers only).	Required for nonIFS providers.	The Ws2_32.dll needs to query and/or modify internal state information associated with the socket handle.
<a href="#">WPUQueueApc</a>	Queues a user-mode APC to the specified thread.	Optional.	The <a href="#">QueueUserApc</a> may also be used.
<a href="#">WPUResetEvent</a>	Resets an event object.	Optional.	The provider may use an appropriate Windows call instead.
<a href="#">WPUSetEvent</a>	Sets an event object.	Optional.	The provider may use an appropriate Windows call instead.

# Installation and Configuration Functions

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following functions are implemented in the Ws2\_32.dll, and are intended to be used by applications that install Windows Sockets transport and namespace service providers on a computer. These functions neither affect currently running applications, nor are the changes made by these functions visible to currently running applications.

For all providers, a provider identifier is a GUID which is generated by the developer of the provider (using the Uuidgen.exe utility) and supplied to Ws2\_32.dll.

FUNCTION	DESCRIPTION
<a href="#">WSCDeinstallProvider</a>	Removes a transport service provider from the registry.
<a href="#">WSCDeinstallProvider32</a>	Removes the specified 32-bit transport service provider from the registry on a 64-bit platform.
<a href="#">WSCEnumProtocols</a>	Retrieves information about available transport protocols.
<a href="#">WSCEnumProtocols32</a>	Retrieves information about available transport protocols in the 32-bit catalog on 64-bit platforms.
<a href="#">WSCInstallProvider</a>	Registers a new transport service provider.
<a href="#">WSCInstallProvider64_32</a>	Registers a new transport service provider into the 32-bit and 64-bit system configuration databases on a 64-bit platform.
<a href="#">WSCInstallProviderAndChains</a>	Registers a new 32-bit transport service provider as well as its specific protocol chains into the system configuration database on a 32-bit platform.
<a href="#">WSCInstallProviderAndChains64_32</a>	Registers a new transport provider and its specific protocol chains into both the 32-bit and 64-bit system configuration databases on a 64-bit platform.

# Namespace Service Providers

3/5/2021 • 3 minutes to read • [Edit Online](#)

A namespace provider implements an interface mapping between the Winsock namespace SPI and the native programmatic interface of an existing name service such as DNS, X.500, or NetWare Directory Services (NDS). While a namespace provider supports exactly one namespace, it is possible for multiple providers for a given namespace to be installed. It is also possible for a single DLL to create an instance of multiple namespace providers. As namespace providers are installed, a catalog of **WSANAMESPACE\_INFO** structures is maintained. An application may use **WSAEnumNameSpaceProviders** to discover which namespaces are supported on a computer.

On Windows Vista and later, an enhanced **WSANAMESPACE\_INFOEX** structure and **WSAEnumNameSpaceProvidersEx** function are provided.

On 64-bit platforms, similar **WSCEnumNameSpaceProviders32** and **WSCEnumNameSpaceProvidersEx32** functions are provided to enumerate the 32-bit catalog.

Refer to [Winsock Namespace Service Provider Requirements](#) for detailed information.

## Legacy GetXbyY Service Providers

Windows Sockets 2 fully supports the TCP/IP-specific name resolution facilities found in Windows Sockets version 1.1. It does this by including the set of **GetXbyY** functions in the SPI. However, the treatment of this set of functions is somewhat different from the rest of the SPI functions. The **GetXbyY** functions appearing in the SPI are prefaced with **GETXBYSP\_**, and are summarized in the following table.

### Berkeley Style Functions

SPI FUNCTION NAME	DESCRIPTION
GETXBYSP_gethostbyaddr	Supplies a <b>hostent</b> structure for the specified host address.
GETXBYSP_gethostbyname	Supplies a <b>hostent</b> structure for the specified host name.
GETXBYSP_getprotobynamne	Supplies a <b>protoent</b> structure for the specified protocol name.
GETXBYSP_getprotobynumber	Supplies a <b>protoent</b> structure for the specified protocol number.
GETXBYSP_getservbyname	Supplies a <b>servent</b> structure for the specified service name.
GETXBYSP_getservbyport	Supplies a <b>servent</b> structure for the service at the specified port.
GETXBYSP_gethostname	Returns the standard host name for the local computer.

### Async Style Functions

SPI FUNCTION NAME	DESCRIPTION
GETXBYYSP_WSAAsyncGetHostByAddr	Supplies a <b>hostent</b> structure for the specified host address.
GETXBYYSP_WSAAsyncGetHostByName	Supplies a <b>hostent</b> structure for the specified host name.
GETXBYYSP_WSAAsyncGetProtoByName	Supplies a <b>protoent</b> structure for the specified protocol name.
GETXBYYSP_WSAAsyncGetProtoByNumber	Supplies a <b>protoent</b> structure for the specified protocol number.
GETXBYYSP_WSAAsyncGetServByName	Supplies a <b>servent</b> structure for the specified service name.
GETXBYYSP_WSAAsyncGetServByPort	Supplies a <b>servent</b> structure for the service at the specified port.
GETXBYYSP_WSACancelAsyncRequest	Cancels an asynchronous <b>GetXbyY</b> operation.

The syntax and semantics of these **GetXbyY** functions in the SPI are exactly the same as those documented in the API Specification and are, therefore, not repeated here.

The Windows Sockets 2 DLL allows exactly one service provider to offer these services. Therefore, there is no need to include pointers to these functions in the procedure table received from service providers at startup. In Windows environments the path to the DLL that implements these functions is retrieved from the value found in the following registry path. This registry entry does not exist by default:

HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Services\WinSock2\Parameters\GetXByYLibraryPath

## Built-In Default GetXbyY Service Provider

A default **GetXbyY** service provider is integrated into the standard Windows Sockets 2 run-time components. This default provider implements all of the above functions, thus it is not required for these functions to be implemented by any namespace provider. However, a namespace provider is free to provide any or all of these functions (and thus override the defaults) by simply storing the string which is the path to the DLL that implements these functions in the indicated registry key. Any of the **GetXbyY** functions not exported by the named provider DLL will be supplied through the built-in defaults. Note, however, that if a provider elects to supply any of the async version of the **GetXbyY** functions, he should supply all of the async functions so that the cancel operation will work appropriately.

The current implementation of the default **GetXbyY** service provider resides within the Wsock32.dll. Depending on how the TCP/IP settings have been established through Control Panel, name resolution will occur using either DNS or local host files. When DNS is used, the default **GetXbyY** service provider uses standard Windows Sockets 1.1 API calls to communicate with the DNS server. These transactions will occur using whatever TCP/IP stack is configured as the default TCP/IP stack. Two special cases however, deserve special mention.

The default implementation of GETXBYYSP\_gethostname obtains the local host name from the registry. This will correspond to the name assigned to "My Computer". The default implementation of GETXBYYSP\_gethostname and GETXBYYSP\_WSAAsyncGetHostByName always compares the supplied host name with the local host name. If they match, the default implementation uses a private interface to probe the Microsoft TCP/IP stack in order to discover its local IP address. Thus, in order to be completely independent of the Microsoft TCP/IP stack, a namespace provider must implement both GETXBYYSP\_gethostname and

GETXBYYSP\_WSAAsyncGetHostByName.

# Name Resolution Model for the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 namespaces.
- The ability of the client application to find the service within a namespace and obtain the required transport protocol and addressing information.

For those accustomed to developing TCP/IP-based applications, this may 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 range of capabilities found in existing name services, the Windows Sockets 2 interface adopts the model described below.

A *namespace* 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 namespaces are currently in wide use including the Internet's Domain Name System (DNS), NetWare Directory Services (NDS), X.500, etc. These namespaces 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 Namespaces in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

There are three different types of namespaces in which a service could be registered:

- Dynamic
- Static
- Persistent

Dynamic namespaces allow services to register with the namespace on the fly, and for clients to discover the available services at run time. Dynamic namespaces frequently rely on broadcasts to indicate the continued availability of a network service. Examples of dynamic namespaces include the SAP namespace used within a NetWare environment and the NBP namespace used by AppleTalk.

Static namespaces require all of the services to be registered ahead of time, that is, when the namespace is created. The DNS is an example of a static namespace. Although there is a programmatic way to resolve names, there is no programmatic way to register names.

Persistent namespaces allow services to register with the namespace on the fly. Unlike dynamic namespaces however, persistent namespaces retain the registration information in nonvolatile storage where it remains until such time as the service requests that it be removed. Persistent namespaces 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.

# Namespace Organization in the SPI

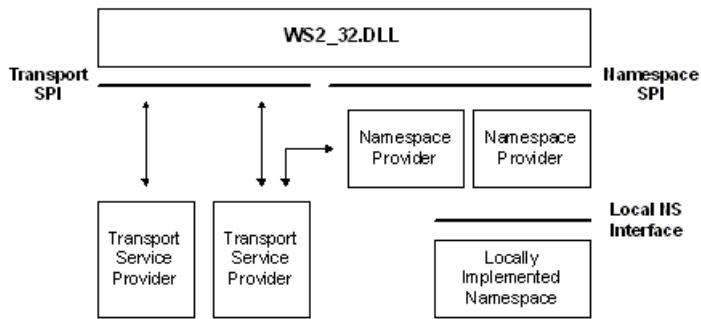
3/5/2021 • 2 minutes to read • [Edit Online](#)

Many namespaces 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 namespace hierarchy from which the search will begin.

# Namespace Provider Architecture in the SPI

3/5/2021 • 3 minutes to read • [Edit Online](#)

Programmatic interfaces used to query the various types of namespaces and to register information within a namespace, if supported, differ widely. A namespace provider is a locally-resident application that can map between the Windows Sockets namespace SPI and some existing namespace that could be implemented locally or accessed through the network. This is illustrated as follows:



## NOTE

It is possible for a given namespace, for example DNS, to have more than one namespace provider installed on a given computer.

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, and so on.

As the example attempts to illustrate, some service classes are well known while others are unique and specific to a particular vertical application. In either case, every service class is represented by both a class name and a class identifier. The class name does not necessarily need to be unique, but the class identifier must be. Globally Unique Identifiers (GUIDs) are used to represent service class IDs. For well-known services, class names, and class identifiers (GUIDs) have been preallocated, and macros are available to convert between, for example, TCP port numbers and the corresponding class identifier GUIDs. For other services, the developer chooses the class name and uses the Uuidgen.exe utility to generate a GUID for the class identifier.

The concept 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 supplied to Windows Sockets at the time the service class is defined, and is referred to as the service class schema information. The Ws2\_32.dll in turn relays this information to all active namespace providers. When an instance of a service is installed and made available on a host computer, its service name is used to distinguish this particular instance from others that may be known to the namespace.

Be aware that the installation of a service class only needs to occur on computers where the service executes, not on all of the clients which may utilize the service. When possible, the Ws2\_32.dll will provide service class schema information to a namespace provider at the time an instance 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 namespace provider that has indicated its ability to supply this data. Because there is no guarantee that the Ws2\_32.dll will be able to supply the service class schema, namespace providers that require this

information must have a fallback mechanism to obtain it through namespace-specific means.

The Internet Domain Name System does not have a well-defined means to store service class schema information. As a result, DNS namespace providers will only be able to accommodate well-known TCP/IP services for which a service class GUID has been preallocated. In practice, this is not a serious limitation since service class GUIDs have been preallocated for the entire set of TCP and UDP ports, and macros are available to retrieve the GUID associated with any TCP or UDP port. Thus, all of the familiar services such as ftp, telnet, whois, etc. are well supported. When querying for these services, by convention the host name of the target computer is the service instance name.

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 computer 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.

# Name resolution for DLL and service providers

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following paragraphs describe how the Ws2\_32.dll and the namespace providers implement the name resolution services supported by the Winsock API.

## Ws2\_32.dll Functionality for Name Resolution

The Ws2\_32.dll manages the registration and demand loading of individual namespace provider DLLs. It also is responsible for routing namespace operations from a Windows Sockets 2 application to the appropriate set of namespace providers. This mapping is governed by the value of namespace and service provider identifier parameters that are found in individual API functions. As a general rule, when a specific namespace provider is referenced, the operation is only routed to an identified provider. If the namespace provider identifier is null but a particular namespace is referenced, the operation is routed to all namespace providers that support the identified namespace. If the namespace provider identifier is null and the namespace identifier is given as NS\_ALL, then the operation is routed to all active namespace providers.

As part of starting a query to one or more service providers, the Ws2\_32.dll allocates an object to keep track of the ongoing state of the query. An opaque handle representing this object is returned to the application that started the query. The application supplies this handle as a parameter each time it repetitively calls an application interface function to retrieve the next unit of data resulting from the query.

In response to these application interface procedure calls, the Ws2\_32.dll uses the information it stores in the object to make corresponding calls to the namespace providers involved in the query. The Ws2\_32.dll updates the information in its object as each successive application interface call occurs so that the corresponding calls to namespace providers progress appropriately through all of the namespace providers involved in the query.

## Namespace Provider Functionality

Each namespace provider is responsible for mapping the set of functions appearing in the Windows Sockets 2 name resolution SPI to the appropriate transactions with the supported namespace. In some cases, this is primarily a matter of mapping the SPI to whatever native interface exists for the namespace. In others, the namespace provider must conduct transactions with the namespace provider over the network. Some namespace providers will do this by making calls to the Windows Sockets API, others will use private interfaces to associated transport stacks.

# Name Resolution Mapping Between API and SPI Functions

3/5/2021 • 2 minutes to read • [Edit Online](#)

The installation of service classes, registration of service instances and basic query operations all map fairly directly from the API to the SPI. The [WSAGetServiceClassNameByClassId](#) function does not have a corresponding function in the SPI, as this function is implemented in Ws2\_32.dll by making a call to [NSPGetServiceClassInfo](#).

The helper functions [WSAAddressToString](#) and [WSAStringToAddress](#) are mapped to the corresponding functions in the transport API, as only a transport provider will necessarily know how to perform the translation on a [sockaddr](#) structure.

# Name Resolution Configuration and Installation

3/5/2021 • 2 minutes to read • [Edit Online](#)

In order for a namespace provider to be accessible through Windows Sockets it must be properly installed on the system and registered with Windows Sockets. When a namespace provider is installed by invoking a vendor's installation program, configuration information must be added to a configuration database to give the Ws2\_32.dll required information regarding the service provider. The Ws2\_32.dll exports

**WSCInstallNameSpace** for the vendor's installation program to use. This function is used to supply relevant information about the service provider to be installed. This information includes:

- Provider Name — A string representing the provider for display in Control Panel.
- Provider Version — The version of this provider.
- Provider Path — A path name to the provider DLL.
- Namespace — The namespace supported by the provider.
- Provider GUID — A unique, vendor-supplied number representing this provider/namespace combination. This is used as a key for all subsequent references to this provider, and for uninstall. These values are created using the Uuidgen.exe utility.
- Stores all flags — a flag indicating whether this namespace provider will be responsible for retaining all service class schema information for all service classes. If such a provider exists, the Ws2\_32.dll does not need to query each individual namespace provider for this information.

On Windows Vista and later, an enhanced **WSCInstallNameSpaceEx32** function is provided that allows the namespace provider to pass an additional blob of data specific to the namespace.

On 64-bit platforms, similar **WSCInstallNameSpace32** and **WSCInstallNameSpaceEx32** functions are provided to install a namespace in the 32-bit catalog.

The Ws2\_32.dll also provides a function, **WSCUnInstallNameSpace**, for a vendor's deinstallation program to remove all the relevant information from the configuration database. The exact location and format of this configuration information is private to the Ws2\_32.dll, and can only be manipulated by the above-mentioned functions.

On 64-bit platforms, a similar **WSCInstallNameSpace32** function is provided to uninstall a namespace in the 32-bit catalog.

At any point, an namespace provider is considered to be either active or inactive, with this setting controlled through the **WSCEnableNSProvider** and **WSCEnableNSProvider32** functions. Namespace providers that are inactive continue to show up when enumerated (using the **WSAEnumNameSpaceProviders**, **WSAEnumNameSpaceProvidersEx**, **WSCEnumNameSpaceProviders32**, and **WSCEnumNameSpaceProvidersEx32** functions), but the Ws2\_32.dll will not route any query or service registration operations to these providers. This capability can be useful in situations where more than one of the installed namespace providers can support a given namespace.

When multiple namespace providers are referenced in a single API function, the order in which the order in which the queries and registration operations are routed to namespace providers is unspecified. The order is unrelated to the order in which namespace providers are installed. There are two ways to control which namespace providers are used to resolve a name query. First, the **WSCEnableNSProvider** and **WSCEnableNSProvider32** functions can be used to enable and disable namespaces in a computer-wide, persistent way. Second, applications can direct an individual query to a particular provider by specifying that provider's identifying GUID as part of the query.



# Winsock Namespace Service Provider Requirements

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following sections provide a description of each of the functional areas that namespace providers are required to implement. Where appropriate, implementation considerations and guidelines are also provided.

- [Summary of Namespace Provider Functions](#)
- [Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 SPI](#)
- [Sample Code for a Service Provider](#)

# Summary of Namespace Provider Functions

3/5/2021 • 2 minutes to read • [Edit Online](#)

The namespace service provider functions can be grouped into five categories:

- Namespace provider configuration (and installation)
- Provider initialization
- Service installation
- Client queries
- Helper functions (and macros)

The following sections identify the functions in each category and briefly describe their intended use. Key data structures are also described.

- [Namespace Provider Configuration and Installation](#)
- [Namespace Provider Initialization and Cleanup](#)
- [Service Installation in the Windows Sockets 2 SPI](#)
- [Service Query](#)
- [Helper Functions in the SPI](#)

# Namespace Provider Configuration and Installation

3/5/2021 • 2 minutes to read • [Edit Online](#)

- [WSCEnableNSProvider](#)
- [WSCEnableNSProvider32](#)
- [WSCInstallNameSpace](#)
- [WSCInstallNameSpace32](#)
- [WSCInstallNameSpaceEx](#)
- [WSCInstallNameSpaceEx32](#)
- [WSCUnInstallNameSpace](#)
- [WSCUnInstallNameSpace32](#)
- [WSCWriteNameSpaceOrder](#)
- [WSCWriteNameSpaceOrder32](#)

As mentioned previously, the installation application for a namespace provider must call [WSCInstallNameSpace](#) or [WSCInstallNameSpaceEx](#) to register with the Ws2\_32.dll and supply static configuration information. To install into the 32-bit catalog on a 64-bit platform, the namespace provider must call [WSCInstallNameSpace32](#) or [WSCInstallNameSpaceEx32](#). The Ws2\_32.dll uses this information to accomplish its routing function and in its implementation of [WSAEnumNameSpaceProviders](#) and [WSAEnumNameSpaceProvidersEx](#). The [WSCUnInstallNameSpace](#) function is used to remove a namespace provider from the registry, and the [WSCEnableNSProvider](#) function is used to toggle a provider between the active and inactive states.

On a 64-bit platform, [WSCUnInstallNameSpace32](#) and [WSCEnableNSProvider32](#) are similar functions to deal with the 32-bit catalog.

The results of these three operations are not visible to applications that are currently loaded and running. Only applications that begin executing after these operations have occurred will be affected by them.

This architecture explicitly supports the instantiation of multiple namespace providers within a single DLL, however each such provider must have a unique namespace provider identifier (GUID) allocated, and a separate call to [WSCInstallNameSpace](#) or [WSCInstallNameSpaceEx](#) must occur for each instantiation (On 64-bit platforms, the functions for the 32-bit catalog are [WSCInstallNameSpace32](#) and [WSCInstallNameSpaceEx32](#)). Such a provider can determine which instantiation is being invoked because the namespace provider (NSP) identifier appears as a parameter in every NSP function.

# Namespace Provider Initialization and Cleanup

3/5/2021 • 2 minutes to read • [Edit Online](#)

- [NSPStartup](#)
- [NSPCleanup](#)

As is the case for the transport SPI, a namespace provider is initialized with a call to [NSPStartup](#) and is terminated with a final call to [NSPCleanup](#). Calls to the startup function may be nested, but will be matched by corresponding calls to the cleanup function. A provider should employ reference counting to determine when the final call to [NSPCleanup](#) has occurred.

# Service Installation in the Windows Sockets 2 SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

- [NSPInstallServiceClass](#)
- [NSPRemoveServiceClass](#)
- [NSPSetService](#)

When the required service class does not already exist, a namespace SPI client uses [NSPInstallServiceClass](#) to install a new service class by supplying a service class name, a GUID for the service class identifier, and a series of [WSANSCLASSINFO](#) structures. These structures are each specific to a particular namespace, and supply common values such as recommended TCP port numbers or NetWare SAP Identifiers. A service class can be removed by calling [NSPRemoveServiceClass](#) and supplying the GUID corresponding to the class identifier.

Once a service class exists, specific instances of a service can be installed or removed via [NSPSetService](#). 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 identifier, service name (for this instance), applicable namespace identifier and protocol information, and a set of transport addresses to which the service listens.

# Service Query

3/5/2021 • 2 minutes to read • [Edit Online](#)

- [NSPLookupServiceBegin](#)
- [NSPLookupServiceNext](#)
- [NSPLookupServiceEnd](#)

A name service query involves a series of calls: [NSPLookupServiceBegin](#), followed by one or more calls to [NSPLookupServiceNext](#) and ending with a call to [NSPLookupServiceEnd](#). [NSPLookupServiceBegin](#) 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 [NSPLookupServiceNext](#) and [NSPLookupServiceEnd](#).

The namespace SPI client invokes [NSPLookupServiceNext](#) to obtain query results, with results supplied in an client-supplied [WSAQUERYSET](#) buffer. The client continues to call [NSPLookupServiceNext](#) 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 [NSPLookupServiceEnd](#). The [NSPLookupServiceEnd](#) function can also be used to cancel a currently pending [NSPLookupServiceNext](#) when called from another thread.

In Windows Sockets 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. Namespace providers should switch to using the [WSA\\_E\\_NO\\_MORE](#) error code as soon as possible to maintain compatibility with the widest possible range of applications.

# Helper Functions in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

- [NSPGetServiceClassInfo](#)

The [NSPGetServiceClassInfo](#) function retrieves service class schema information that has been retained by a namespace provider. It is also used by the Windows Sockets 2 DLL in its implementation of [WSAGetServiceClassNameByClassId](#).

The following macros defined in the *Svcguid.h* header file and can aid in mapping between well known service classes and these namespaces.

MACRO NAME	DESCRIPTION
SVCID_TCP(Port) SVCID_UDP(Port)	Given a TCP or UDP port for the Internet protocol, returns the GUID.
IS_SVCID_TCP(GUID) IS_SVCID_UDP(GUID)	Returns TRUE if the GUID for TCP or UDP is within the allowable range.
PORT_FROM_SVCID_TCP(GUID) PORT_FROM_SVCID_UDP(GUID)	Returns the TCP or UDP port associated with the GUID.
SVCID_NETWARE(SAPID)	Given the Service Advertising Protocol (SAP) identifier, returns the GUID. This macro is used with the SAP namespace within a NetWare environment.
SAPID_FROM_SVCID_NETWARE(GUID)	Returns the NetWare SAP identifier associated with the GUID. This macro is used with the SAP namespace within a NetWare environment.
IS_SVCID_NETWARE(GUID)	Returns TRUE if the GUID for NetWare is within the allowable range. This macro is used with the SAP namespace within a NetWare environment.

**NOTE**

The *Svcguid.h* header file is not automatically included by the *Winsock2.h* header file.

# Name Resolution Data Structures in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

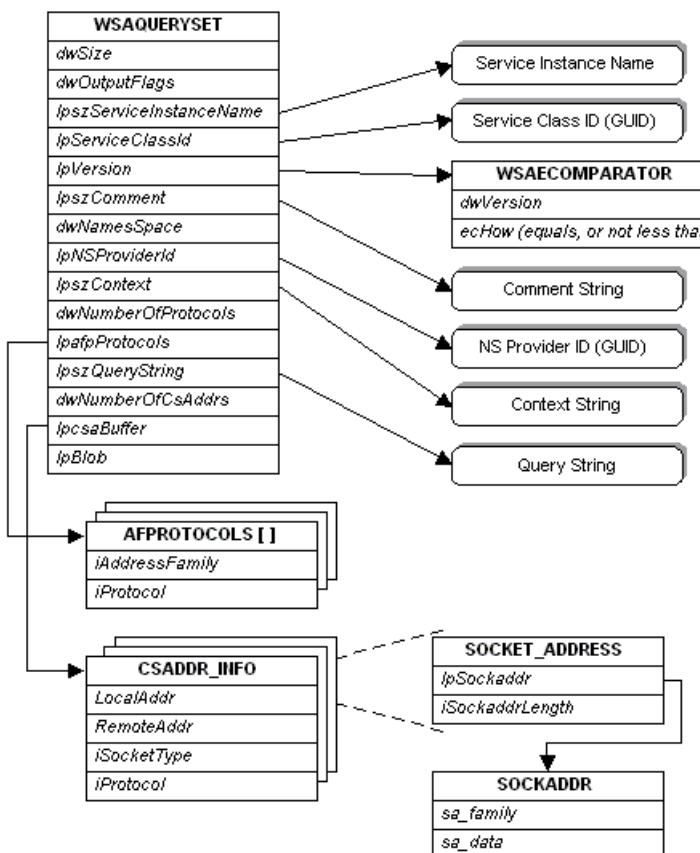
There are several important data structures that are used extensively throughout the name resolution functions. These are described below.

- [Query-Related Data Structures in the SPI](#)
- [Service Class Data Structures in the SPI](#)

# Query-Related Data Structures in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

The **WSAQUERYSET** structure is used to form queries for **NSPLookupServiceBegin**, and used to deliver query results for **NSPLookupServiceNext**. It is a complex structure because 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:



Within the **WSAQUERYSET** structure, most of the members are self explanatory, but some deserve additional explanation. The **dwSize** will be filled in with `sizeof( WSAQUERYSET )`, and can be used by namespace providers to detect and adapt to different versions of the **WSAQUERYSET** structure that may appear.

The **dwOutputFlags** member is used by a namespace provider to provide additional data about query results. For more information, see **NSPLookupServiceNext**.

The **WSAECOMPARATOR** structure referenced by **lpVersion** is used for both query constraint and results. For queries, the **dwVersion** member indicates the desired version of the service. The **ecHow** member is an enumerated type which specifies how the comparison will be made. The choices are **COMP\_EQUALS** which requires that an exact match in version occurs, or **COMP\_NOTLESS** which specifies that the service's version number be no less than the value of **dwVersion**.

The interpretation of **dwNameSpace** and **lpNSProviderId** depends upon how the structure is being used and is described further in the individual function descriptions that utilize this structure.

The **lpszContext** member applies to hierarchical namespaces, 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 namespace.

- 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 namespace specific. If **dwNameSpace** is NS\_ALL, then only "" or "\" should be passed as the context because these are recognized by all namespaces.

The **IpszQueryString** member is used to supply additional, namespace-specific query information such as a string describing a well-known service and transport protocol name, as in "ftp/tcp".

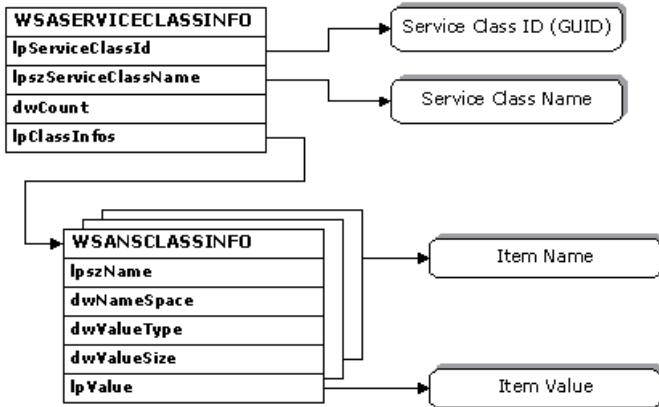
The **AFPROTOCOLS** structure referenced by **IpafpProtocols** is used for query purposes only, and supplies a list of protocols to constrain the query. These protocols are represented as (address family, protocol) pairs, because protocol values only have meaning within the context of an address family.

The array of **CSADDR\_INFO** structure referenced by **IpcaBuffer** contains all of the information required 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** members both directly contain a **SOCKET\_ADDRESS** structure. A service would create a socket using the tuple (**LocalAddr.IpSockaddr->sa\_family**, **iSocketType**, **iProtocol**). It would bind the socket to a local address using **LocalAddr.IpSockaddr**, and **LocalAddr.IpSockaddrLength**. The client creates its socket with the tuple (**RemoteAddr.IpSockaddr->sa\_family**, **iSocketType**, **iProtocol**), and uses the combination of **RemoteAddr.IpSockaddr**, and **RemoteAddr.IpSockaddrLength** when making a remote connection.

# Service Class Data Structures in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

When a new service class is installed, a [WSASERVICECLASSINFO](#) structure must be prepared and supplied. This structure also consists of substructures that contain a series of parameters that apply to specific namespaces.



For each service class, there is a single [WSASERVICECLASSINFO](#) structure. Within the [WSASERVICECLASSINFO](#) structure, the service class's unique identifier is contained in **lpServiceClassId**, and an associated display string is referenced by **lpServiceClassName**.

The **lpClassInfos** member in the [WSASERVICECLASSINFO](#) structure references an array of [WSANSCLASSINFO](#) structures, each of which supplies a named and typed parameter that applies to a specified namespace. Examples of values for the **lpszName** member include: SAPIID, TCPPORT, UDPPORT, etc. These strings are generally specific to the namespace identified in **dwNameSpace**. Typical values for **dwValueType** might be REG\_DWORD, REG\_SZ, etc. The **dwValueSize** member 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 namespace provider via [NSPInstallServiceClass](#). Each individual namespace provider then sifts through the list of [WSANSCLASSINFO](#) structures and retain the information applicable to it. This architecture also envisions the future existence of a special namespace provider that would retain all of the service class schema information for all of the namespaces. The Ws2\_32.dll would query this provider to obtain the [WSASERVICECLASSINFO](#) data needed to supply to namespace providers when [NSPLookupServiceBegin](#) is invoked to initiate a query, and when [NSPSetService](#) is invoked to register a service. Namespace provider should not rely on this capability for the time being, and should instead have a provider-specific means to obtain any needed service class schema information. In the absence of a provider that stores all service class schema for all namespaces, the Ws2\_32.dll will use [NSPGetServiceClassInfo](#) to obtain such information from each individual namespace provider.

# Compatible Name Resolution for TCP/IP in the Windows Sockets 1.1 SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

Windows Sockets 1.1 defined a number of routines that were used for IPv4 name resolution with TCP/IP networks. These are customarily called the **GetXbyY** functions and include the following.

[gethostname](#)

[gethostbyaddr](#)

[gethostbyname](#)

[getprotobynam](#)

[getprotobynumber](#)

[getservbyname](#)

[getservbyport](#)

Asynchronous versions of these functions were also defined.

[WSAAAsyncGetHostByAddr](#)

[WSAAAsyncGetHostByName](#)

[WSAAAsyncGetProtoByName](#)

[WSAAAsyncGetProtoByNumber](#)

[WSAAAsyncGetServByName](#)

[WSAAAsyncGetServByPort](#)

These functions are specific to TCP/IP networks; developers of protocol-independent applications are discouraged from continuing to utilize these transport-specific functions. However, to retain strict backward compatibility with Windows Sockets 1.1, the preceding functions continue to be supported as long as at least one namespace provider is present that supports the AF\_INET address family.

The *Ws2\_32.dll* implements these compatibility functions in terms of the new, protocol-independent name resolution facilities using an appropriate sequence of [WSALookupServiceBegin](#), [WSALookupServiceNext](#), [WSALookupServiceEnd](#) 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 that only the implementation of the synchronous **GetXbyY** functions are discussed.

# Basic Approach for GetXbyY in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

Most `GetXbyY` functions are translated by `Ws2_32.dll` to a `WSALookupServiceBegin`, `WSALookupServiceNext`, `WSALookupServiceEnd` 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 members must, of course, be completely contained within the blob, and all strings are ASCII.

# getprotobynumber and getprotobyname Functions in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

These functions are implemented within Ws2\_32.dll by consulting a local protocols database. They do not result in any name resolution query.

# getservbyname and getservbyport Functions in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

The [WSALookupServiceBegin](#) query uses SVCID\_INET\_SERVICEBYNAME as the service class GUID. The *lpszServiceInstanceName* parameter references a string that 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** structure in the blob (using offsets instead of pointers as described above). NSPs should honor these other LUP\_RETURN\_\* flags as well.

FLAG	DESCRIPTION
LUP_RETURN_NAME	Returns the <b>s_name</b> member from <b>SERVENT</b> structure in <i>lpszServiceInstanceName</i> .
LUP_RETURN_TYPE	Returns canonical GUID in <i>lpServiceClassId</i> . It is understood that a service identified as ftp or 21 may be on another port according to locally established conventions. The <b>s_port</b> member of the <b>SERVENT</b> structure should indicate where the service can be contacted in the local environment. The canonical GUID returned when LUP_RETURN_TYPE is set should be one of the predefined GUIDs from svcs.h that corresponds to the port number indicated in the <b>SERVENT</b> structure.

# gethostbyname Function in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

The [WSALookupServiceBegin](#) query uses SVCID\_INET\_HOSTADDRBYNAME as the service class GUID. The host name is supplied in */pszServiceInstanceName*. The *Ws2\_32.dll* specifies LUP\_RETURN\_BLOB and the NSP places a [HOSTENT](#) structure in the blob (using offsets instead of pointers as described above). NSPs should honor these other LUP\_RETURN\_\* flags as well.

FLAG	DESCRIPTION
LUP_RETURN_NAME	Returns the <b>h_name</b> member from <a href="#">HOSTENT</a> structure in <i>/pszServiceInstanceName</i> .
LUP_RETURN_ADDR	Returns addressing information from <a href="#">HOSTENT</a> in <a href="#">CSADDR_INFO</a> structures, port information is defaulted to zero. Note that this routine does not resolve host names consisting of a dotted-decimal IPv4 address string.

# gethostbyaddr Function in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

The [WSALookupServiceBegin](#) query uses SVCID\_INET\_HOSTNAMEBYADDR as the service class GUID. The host address is supplied in *lpszServiceInstanceName* as a dotted-decimal IPv4 address string (for example, 192.9.200.120). The *Ws2\_32.dll* specifies LUP\_RETURN\_BLOB and the NSP places a **HOSTENT** structure in the blob (using offsets instead of pointers as described above). NSPs should honor these other LUP\_RETURN\_\* flags as well.

FLAG	DESCRIPTION
LUP_RETURN_NAME	Returns the <b>h_name</b> member from <b>HOSTENT</b> structure in <i>lpszServiceInstanceName</i> .
LUP_RETURN_ADDR	Returns addressing information from <b>HOSTENT</b> in <b>CSADDR_INFO</b> structures, port information is defaulted to zero.

# gethostname Function in the SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

The [WSALookupServiceBegin](#) query uses SVCID\_HOSTNAME as the service class GUID. If *lpszServiceInstanceName* is null or references a null string (that is . ""), the local host is to be resolved. Otherwise, a lookup on a specified host name shall occur. For the purposes of emulating [gethostname](#) the Ws2\_32.dll will specify a null pointer for *lpszServiceInstanceName*, and specify LUP\_RETURN\_NAME so that the host name is returned in the *lpszServiceInstanceName* parameter. If an application uses this query and specifies LUP\_RETURN\_ADDR then the host address will be provided in a CSADDR\_INFO structure. The LUP\_RETURN\_BLOB action is undefined for this query. Port information will be defaulted to zero unless the *lpszQueryString* references a service such as ftp, in which case the complete transport address of the indicated service will be supplied.

# Sample Code for a Service Provider

3/5/2021 • 9 minutes to read • [Edit Online](#)

This section contains a source code sample that demonstrates how to implement the **GetXbyY** functions using the new, protocol-independent RNR functions. A developer should implement these functions as a starting point. To comply with the Windows Sockets specification, many more functions are needed.

## IMPORTANT

The following code is not guaranteed to compile.

```
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <windns.h>
#include <svccguid.h>
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

#undef WINSOCK_API_LINKAGE

#define WINSOCK_API_LINKAGE __control_entrypoint(DllExport)

/*++
xbyrnr.cpp

GetXbyY emulation via new WinSock2 RNR. This source module shows
code that is built into the WinSock2 DLL (ws2_32.dll). It
demonstrates how the older GetXByY functions are mapped to the new
WSALookupServiceBegin, WSALookupServiceNext, WSALookupServiceEnd
functions.

This module is not guaranteed to compile. It is provided as source
code for RNR namespace service providers to understand what will
be coming down to their code in response to the traditional
GetXbyY calls.

At this time, only
    gethostname
    gethostbyname
    gethostbyaddr
    getservbyname
    getservbyport
are implemented in this manner.

Warning: This is not provided as a template for either RNR
applications or namespace providers. This code is only intended
to illustrate what happens in the WinSock2 DLL to map the GetXbyY
calls to the new RNR APIs.

--*/
```

```

// The initial buffer size passed by getxxyy functions
// to WSALookupServiceNext. If it is insufficient for the query,
// the amount specified by the provider is allocated and call is
// repeated.

// The initial buffer is allocated from stack, so we try to keep it
// relatively small, but still for performance reasons we want to be able
// to satisfy most of the calls with just this amount

#define RNR_BUFFER_SIZE (sizeof(WSAQUERYSET) + 256)

//
// Forward declares
//

LPBLOB getxyDataEnt(
    PCHAR *pResults,
    DWORD dwSize,
    LPSTR lpszName,
    LPGUID lpType,
    LPSTR *lppName
);

VOID FixList(PCHAR ** List, PCHAR Base);

VOID UnpackHostEnt(struct hostent * hostent);

VOID UnpackServEnt(struct servent * servent);

GUID HostAddrByNameGuid = SVCID_INET_HOSTADDRBYNAME;
GUID HostNameGuid = SVCID_HOSTNAME;
GUID AddressGuid = SVCID_INET_HOSTADDRBYINETSTRING;
GUID IANAGuid      = SVCID_INET_SERVICEBYNAME;

//
// Utility to turn a list of offsets into a list of addresses. Used
// to convert structures returned as BLOBS.
//

VOID FixList(PCHAR ** List, PCHAR Base)
{
    if(*List)
    {
        PCHAR * Addr;

        Addr = *List = (PCHAR *)((DWORD)*List + Base);
        while(*Addr)
        {
            *Addr = (PCHAR)((DWORD)*Addr + Base));
            Addr++;
        }
    }
}

//
// Routine to convert a hostent returned in a BLOB to one with
// usable pointers. The structure is converted in-place.
//
VOID UnpackHostEnt(struct hostent * hostent)
{
    PCHAR pch;

    pch = (PCHAR)hostent;

    if(hostent->h_name)
    {
        hostent->h_name = (PCHAR)((DWORD)hostent->h_name + pch);
    }
}

```

```

    }
    FixList(&hostent->h_aliases, pch);
    FixList(&hostent->h_addr_list, pch);
}

//  

// Routine to unpack a servent returned in a BLOB to one with  

// usable pointers. The structure is converted in-place  

//  

VOID UnpackServEnt(struct servent * servent)
{
    PCHAR pch;

    pch = (PCHAR)servent;

    FixList(&servent->s_aliases, pch);
    servent->s_name = (PCHAR)(DWORD(servent->s_name) + pch);
    servent->s_proto = (PCHAR)(DWORD(servent->s_proto) + pch);
}

WINSOCK_API_LINKAGE
struct hostent FAR *
WSAAPI
gethostbyaddr(
    IN const char *addr,
    IN int len,
    IN int type
)
/*++  

Routine Description:  


```

Get host information corresponding to an address.

#### Arguments:

addr - A pointer to an address in network byte order.

len - The length of the address, which must be 4 for PF\_INET addresses.

type - The type of the address, which must be PF\_INET.

#### Returns:

If no error occurs, gethostbyaddr() returns a pointer to the hostent structure described above. Otherwise it returns a NULL pointer and a specific error code is stored with SetErrorCode().

```
--*/  

{  

    CHAR qbuf[DNS_MAX_NAME_BUFFER_LENGTH];  

    struct hostent *ph;  

    LPBLOB pBlob;  

    PCHAR pResults;  

    CHAR localResults[RNR_BUFFER_SIZE];  

    INT ErrorCode;  

    PTHREAD Thread;  

    ErrorCode = PROLOG(&Thread);  

    if(ErrorCode != NO_ERROR)  

    {  

        SetLastError(ErrorCode);  

        return(NULL);  

    }  

    if ( !addr )  

    {  

        SetLastError(WSAEINVAL);  

        return(NULL);  

    }
```

```

}

pResults = localResults;

//
// NOTICE. Only handles current inet address forms.
//
(void)sprintf_s(qbuf, sizeof(qbuf), "%u.%u.%u.%u",
    ((unsigned)addr[0] & 0xff),
    ((unsigned)addr[1] & 0xff),
    ((unsigned)addr[2] & 0xff),
    ((unsigned)addr[3] & 0xff));

pBlob = getxyDataEnt(pResults,
    RNR_BUFFER_SIZE,
    qbuf,
    &AddressGuid,
    0);
if(pBlob)
{
    ph = (struct hostent *)Thread->CopyHostEnt(pBlob);
    if(ph)
    {
        UnpackHostEnt(ph);
    }
}
else
{
    ph = 0;
    if(GetLastError() == WSASERVICE_NOT_FOUND)
    {
        SetLastError(WSANO_ADDRESS);
    }
}
if (pResults != localResults)
    delete pResults;

return(ph);
} // gethostbyaddr

```

#### WINSOCK\_API\_LINKAGE

```

struct hostent* FAR WSAAPI
gethostbyname(
    IN const char FAR * name
)
/*++
Routine Description:
```

Get host information corresponding to a hostname.

#### Arguments:

name - A pointer to the null-terminated name of the host.

#### Returns:

If no error occurs, gethostbyname() returns a pointer to the hostent structure described above. Otherwise it returns a null pointer and a specific error code is stored with SetErrorCode().

```
--*/
{
    struct hostent * hent;
    LPBLOB pBlob;
    PCHAR pResults;
    CHAR localResults[RNR_BUFFER_SIZE];
    INT ErrorCode;
    CHAR szLocalName[200]; // for storing the local name. This
```

```

        // is simply a big number assumed
        // to be large enough. This is used
        // only when the caller chooses not to
        // provide a name.

PCHAR pszName;

PDTHREAD Thread;

ErrorCode = PROLOG(&Thread);
if(ErrorCode != NO_ERROR)
{
    SetLastErrorMessage(ErrorCode);
    return(NULL);
}

/*
// A NULL input name means look for the local name. So,
// get it.
//
if(!name || !*name)
{
    if(gethostname(szLocalName, 200) != NO_ERROR)
    {
        return(NULL);
    }
    pszName = szLocalName;
}
else
{
    pszName = (PCHAR)name;
}

pResults = localResults;

pBlob = getxyDataEnt( &pResults,
                      RNR_BUFFER_SIZE,
                      pszName,
                      &HostAddrByNameGuid,
                      0);

if(pBlob &&
   ( !name || !*name ) )
{
    pBlob = getxyDataEnt( &pResults,
                          RNR_BUFFER_SIZE,
                          NULL,
                          &HostAddrByNameGuid,
                          0);
}

if(pBlob )
{
    hent = (struct hostent *)Thread->CopyHostEnt(pBlob);
    if(hent)
    {
        UnpackHostEnt(hent);
    }
}
else
{
    hent = 0;
    if(GetLastError() == WSASERVICE_NOT_FOUND)
    {
        SetLastError(WSAHOST_NOT_FOUND);
    }
}

if (pResults!=localResults)
    delete pResults;

return(hent);

```

```

} // gethostname

WINSOCK_API_LINKAGE
int WSAAPI
gethostname(
    OUT char FAR * name,
    IN int namelen
)
/*++
Routine Description:

    Return the standard host name for the local computer.

Arguments:

    name      - A pointer to a buffer that will receive the host name.

    namelen - The length of the buffer.

Returns:

    Zero on success else SOCKET_ERROR. The error code is stored with
    SetErrorCode().
--*/
{
    PCHAR lpName;
    PCHAR pResults;
    CHAR localResults[RNR_BUFFER_SIZE];
    INT ErrorCode;
    PDTHREAD Thread;

    ErrorCode = PROLOG(&Thread);
    if(ErrorCode != NO_ERROR)
    {
        gSetLastError(ErrorCode);
        return(NULL);
    }

    pResults = localResults;

    if(getxyDataEnt(pResults,
                    RNR_BUFFER_SIZE,
                    NULL,
                    &HostNameGuid,
                    &lpName
                    ))
    {
        INT iSize = strlen(lpName) + 1;

        if(iSize <= namelen)
        {
            memcpy(name, lpName, iSize);
        }
        else
        {
            SetLastError(WSAEFAULT);
            ErrorCode = SOCKET_ERROR;
        }
    }
    else
    {
        ErrorCode = SOCKET_ERROR; // assume LastError has been set
    }

    if (pResults!=localResults)
        delete pResults;

    return(ErrorCode);
} // gethostname

```

```

j /> _getservbyport

WINSOCK_API_LINKAGE
struct servent FAR * WSAAPI
getservbyport(
    IN int port,
    IN const char FAR * proto
)
/*++
Routine Description:

    Get service information corresponding to a port and protocol.

Arguments:

    port - The port for a service, in network byte order.

    proto - An optional pointer to a protocol name. If this is NULL,
            getservbyport() returns the first service entry for which
            the port matches the s_port. Otherwise getservbyport()
            matches both the port and the proto.

Returns:

    If no error occurs, getservbyport() returns a pointer to the
    servent structure described above. Otherwise it returns a NULL
    pointer and a specific error code is stored with SetErrorCode().
--*/
{
    PCHAR pszTemp;
    struct servent * sent;
    LPBLOB pBlob;
    PCHAR pResults;
    CHAR localResults[RNR_BUFFER_SIZE];
    INT ErrorCode;
    PDTHREAD Thread;

    ErrorCode = PROLOG(&Thread);
    if(ErrorCode != NO_ERROR)
    {
        SetLastError(ErrorCode);
        return(NULL);
    }

    pResults = localResults;

    if(!proto)
    {
        proto = "";
    }

    //
    // the 5 is the max number of digits in a port
    //
    pszTemp = new CHAR[strlen(proto) + 1 + 1 + 5];
    sprintf_s(pszTemp, strlen(proto)+1+1+5, "%d/%s", (port & 0xffff), proto);
    pBlob = getxyDataEnt(pResults,
                         RNR_BUFFER_SIZE,
                         pszTemp,
                         &IANAGuid,
                         0
                         );
    delete pszTemp;
    if(!pBlob)
    {
        sent = 0;
        if(GetLastError() == WSATYPE_NOT_FOUND)
        {
            SetLastErron(WSCANO_DATA);
        }
    }
}

```

```

        SetLastErrort(WSANO_DATA);
    }
}
else
{
    sent = (struct servent *)Thread->CopyServEnt(pBlob);
    if(sent)
    {
        UnpackServEnt(sent);
    }
}

if (pResults!=localResults)
    delete pResults;

return(sent);
} // getservbyport

WINSOCK_API_LINKAGE
struct servent FAR * WSAAPI
getservbyname(
    IN const char FAR * name,
    IN const char FAR * proto
)
/*++
Routine Description:

Get service information corresponding to a service name and
protocol.

```

**Arguments:**

name - A pointer to a null-terminated service name.

proto - An optional pointer to a null-terminated protocol name. If this pointer is NULL, getservbyname() returns the first service entry for which the name matches the s\_name or one of the s\_aliases. Otherwise getservbyname() matches both the name and the proto.

**Returns:**

If no error occurs, getservbyname() returns a pointer to the servent structure described above. Otherwise it returns a NULL pointer and a specific error code is stored with SetErrorCode().

```
--*/
{

PCHAR pszTemp;
struct servent * sent;
LPBLOB pBlob;
PCHAR pResults;
CHAR localResults[RNR_BUFFER_SIZE];
INT ErrorCode;
PDTHREAD Thread;

ErrorCode = PROLOG(&Thread);
if(ErrorCode != NO_ERROR)
{
    SetLastError(ErrorCode);
    return(NULL);
}

pResults = localResults;

if(!proto)
{
    proto = "";
}
```

```

pszTemp = new CHAR[strlen(name) + strlen(proto) + 1 + 1];
sprintf_s(pszTemp, strlen(name)+strlen(proto)+1+1, "%s/%s", name, proto);
pBlob = getxyDataEnt(pResults,
                      RNR_BUFFER_SIZE,
                      pszTemp,
                      &IANAGuid,
                      0
                     );
delete pszTemp;
if(!pBlob)
{
    sent = 0;
    if(GetLastError() == WSATYPE_NOT_FOUND)
    {
        SetLastError(WSANO_DATA);
    }
}
else
{
    sent = (struct servent *)Thread->CopyServEnt(pBlob);
    if(sent)
    {
        UnpackServEnt(sent);
    }
}

if (pResults!=localResults)
    delete pResults;

return(sent);
} // getservbyname

// Common routine for obtaining a xxxent buffer. Input is used to
// execute the WSALookup series of APIs.
//
// Args:
//   pResults -- a buffer supplied by the caller to be used in
//             the WSALookup calls. This should be as large as
//             the caller can afford to offer.
//   dwLength -- number of bytes in pResults
//   lpszName -- pointer to the service name. May be NULL
//   lpType   -- pointer to the service type . This should be one of
//             the SVCID_INET_xxxxx types. It may be anything
//             that produces a BLOB.
//   lppName  -- pointer to pointer where the resulting name pointer
//             is stored. May be NULL if the name is not needed.
//
// Returns:
//   0 -- No BLOB data was returned. In general, this means the
//       operation failed. Even if the WSALookupNext succeeded
//       and returned a name, the name will not be returned.
//   else -- a pointer to the BLOB.
//
//
// The protocol restrictions list for all emulation operations. This
// should limit the invoked providers to the set that know about
// hostents and servents. If not, then the special SVCID_INET GUIDs
// should take care of the remainder.
//
AFPROTOCOLS afp[2] = {
    {AF_INET, IPPROTO_UDP},
    {AF_INET, IPPROTO_TCP}
};

LPBLOB
getxyDataEnt(
    PCHAR *pResults,

```

```

    DWORD dwLength,
    LPSTR lpszName,
    LPGUID lpType,
    LPSTR *lppName)
{

    PWSAQUERYSETA pwsaq = (PWSAQUERYSETA)pResults;
    int err;
    HANDLE hRnR;
    LPBLOB pvRet = 0;
    INT Err = 0;

    //
    // create the query
    //
    memset(pwsaq, 0, sizeof(*pwsaq));
    pwsaq->dwSize = sizeof(*pwsaq);
    pwsaq->lpszServiceInstanceId = lpszName;
    pwsaq->lpServiceClassId = lpType;
    pwsaq->dwNameSpace = NS_ALL;
    pwsaq->dwNumberOfProtocols = 2;
    pwsaq->lpafpProtocols = &afp[0];

    err = WSALookupServiceBeginA(pwsaq,
                                LUP_RETURN_BLOB | LUP_RETURN_NAME,
                                &hRnR);

    if(err == NO_ERROR)
    {
        //
        // The query was accepted, so execute it via the Next call.
        //
        err = WSALookupServiceNextA(
            hRnR,
            0,
            &dwLength,
            pwsaq);
        //

        // if NO_ERROR was returned and a BLOB is present, this
        // worked, just return the requested information. Otherwise,
        // invent an error or capture the transmitted one.
        //

        if(err == NO_ERROR)
        {
            if(pvRet = pwsaq->lpBlob)
            {
                if(lppName)
                {
                    *lppName = pwsaq->lpszServiceInstanceId;
                }
            }
            else
            {
                err = WSANO_DATA;
            }
        }
        else
        {
            //
            // WSALookupServiceEnd clobbers LastError so save
            // it before closing the handle.
            //

            err = GetLastError();
        }
        WSALookupServiceEnd(hRnR);

        //
    }
}

```

```
// if an error happened, stash the value in LastError
//  
  
if(err != NO_ERROR)  
{  
    SetLastError(err);  
}  
}  
return(pvRet);  
}
```

# Using Winsock

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section describes procedures and programming techniques employed with Winsock. It includes basic Winsock programming techniques, such as [Getting Started With Winsock](#), as well as advanced techniques useful for experienced Winsock developers.

The following list describes the topics in this section:

- [Getting Started With Winsock](#)
- [Secure Winsock Programming](#)
- [Porting Socket Applications to Winsock](#)
- [IPv6 Guide for Windows Sockets Applications](#)
- [High-performance Windows Sockets Applications](#)
- [Categorizing Layered Service Providers and Applications](#)
- [Multicast Programming](#)
- [Reliable Multicast Programming \(PGM\)](#)
- [Winsock Tracing](#)

## Related topics

[About Winsock](#)

[What's New for Windows Sockets](#)

[Winsock Network Protocol Support in Windows](#)

[Winsock Reference](#)

# Getting Started with Winsock

3/5/2021 • 3 minutes to read • [Edit Online](#)

The following is a step-by-step guide to getting started with Windows Sockets programming. It is designed to provide an understanding of basic Winsock functions and data structures, and how they work together.

The client and server application that is used for illustration is a very basic client and server. More advanced code examples are included in the samples included with the Microsoft Windows Software Development Kit (SDK).

The first few steps are the same for both client and server applications.

- [About Servers and Clients](#)
- [Creating a Basic Winsock Application](#)
- [Initializing Winsock](#)

The following sections describe the remaining steps for creating a Winsock client application.

- [Creating a Socket for the Client](#)
- [Connecting to a Socket](#)
- [Sending and Receiving Data on the Client](#)
- [Disconnecting the Client](#)

The following sections describe the remaining steps for creating a Winsock server application.

- [Creating a Socket for the Server](#)
- [Binding a Socket](#)
- [Listening on a Socket](#)
- [Accepting a Connection](#)
- [Receiving and Sending Data on the Server](#)
- [Disconnecting the Server](#)

The complete source code for these basic examples.

- [Running the Winsock Client and Server Code Sample](#)
- [Complete Winsock Client Code](#)
- [Complete Winsock Server Code](#)

## Advanced Winsock Samples

Several more advanced Winsock client and server samples are included with the Windows SDK. By default, the Winsock sample source code is installed in the following directory by the Windows SDK for Windows 7:

*C:\Program Files\Microsoft SDKs\Windows\v7.0\Samples\NetDs\winsock*

On earlier versions of the Windows SDK, the version number in the above path would change. For example, the Winsock sample source code is installed in the following default directory by the Windows SDK for Windows Vista

*C:\Program Files\Microsoft SDKs\Windows\v6.0\Samples\NetDs\winsock*

The more advanced samples listed below in order from higher to lower performance are found in the following directories:

- iocp

This directory contains three sample programs that use I/O completion ports. The programs include a Winsock server (iocpserver) that uses the [WSAAccept](#) function, a Winsock server (iocpserverex) that uses the [AcceptEx](#) function, and a simple multithreaded Winsock client (iocpclient) used to test either of these servers. The server programs support multiple clients connecting via TCP/IP and sending arbitrary sized data buffers which the server then echoes back to the client. For convenience, a simple client program, iocpclient, was developed to connect and continually send data to the server to stress it using multiple threads. Winsock servers that use I/O completion ports provide the most performance capability.

- overlap

This directory contains a sample server program that uses overlapped I/O. The sample program uses the [AcceptEx](#) function and overlapped I/O to handle multiple asynchronous connection requests from clients effectively. The server uses the [AcceptEx](#) function to multiplex different client connections in a single-threaded Win32 application. Using overlapped I/O allows for greater scalability.

- WSAPoll

This directory contains a basic sample program that demonstrates the use of the [WSAPoll](#) function. The combined client and server program are non-blocking and use the [WSAPoll](#) function to determine when it is possible to send or receive without blocking. This sample is more for illustration and is not a high-performance server.

- simple

This directory contains three basic sample programs that demonstrate the use of multiple threads by a server. The programs include a simple TCP/UDP server (simples), a TCP-only server (simples\_ioctl) that uses the [select](#) function in a Win32 console application to support multiple client requests, and a client TCP/UDP program (simplec) for testing the servers. The servers demonstrate the use of multiple threads to handle multiple client requests. This method has scalability issues since a separate thread is created for each client request.

- accept

This directory contains a basic sample server and client program. The server demonstrates the use of either non-blocking accept using the [select](#) function or asynchronous accept using the [WSAAAsyncSelect](#) function. This sample is more for illustration and is not a high-performance server.

# About Servers and Clients

3/5/2021 • 2 minutes to read • [Edit Online](#)

There are two distinct types of socket network applications: Server and Client.

Servers and Clients have different behaviors; therefore, the process of creating them is different. What follows is the general model for creating a streaming TCP/IP Server and Client.

## Server

1. Initialize Winsock.
2. Create a socket.
3. Bind the socket.
4. Listen on the socket for a client.
5. Accept a connection from a client.
6. Receive and send data.
7. Disconnect.

## Client

1. Initialize Winsock.
2. Create a socket.
3. Connect to the server.
4. Send and receive data.
5. Disconnect.

### NOTE

Some of the steps are the same for a client and a server. These steps are implemented almost exactly alike. Some of the steps in this guide will be specific to the type of application being created.

First Step: [Creating a Basic Winsock Application](#)

## Related topics

[Getting Started With Winsock](#)

# Creating a Basic Winsock Application

3/5/2021 • 2 minutes to read • [Edit Online](#)

## To create a basic Winsock application

1. Create a new empty project.
2. Add an empty C++ source file to the project.
3. Ensure that the build environment refers to the Include, Lib, and Src directories of the Microsoft Windows Software Development Kit (SDK) or the earlier Platform Software Development Kit (SDK).
4. Ensure that the build environment links to the Winsock Library file Ws2\_32.lib. Applications that use Winsock must be linked with the Ws2\_32.lib library file. The #pragma comment indicates to the linker that the *Ws2\_32.lib* file is needed.
5. Begin programming the Winsock application. Use the Winsock API by including the Winsock 2 header files. The *Winsock2.h* header file contains most of the Winsock functions, structures, and definitions. The *Ws2tcpip.h* header file contains definitions introduced in the WinSock 2 Protocol-Specific Annex document for TCP/IP that includes newer functions and structures used to retrieve IP addresses.

### NOTE

Stdio.h is used for standard input and output, specifically the printf() function.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

#pragma comment(lib, "Ws2_32.lib")

int main() {
    return 0;
}
```

### NOTE

The *Iphlpapi.h* header file is required if an application is using the IP Helper APIs. When the *Iphlpapi.h* header file is required, the #include line for the *Winsock2.h* header file should be placed before the #include line for the *Iphlpapi.h* header file.

The *Winsock2.h* header file internally includes core elements from the *Windows.h* header file, so there is not usually an #include line for the *Windows.h* header file in Winsock applications. If an #include line is needed for the *Windows.h* header file, this should be preceded with the #define WIN32\_LEAN\_AND\_MEAN macro. For historical reasons, the *Windows.h* header defaults to including the *Winsock.h* header file for Windows Sockets 1.1. The declarations in the *Winsock.h* header file will conflict with the declarations in the *Winsock2.h* header file required by Windows Sockets 2.0. The WIN32\_LEAN\_AND\_MEAN macro prevents the *Winsock.h* from being included by the *Windows.h* header. An example illustrating this is shown below.

```
#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <iphlpapi.h>
#include <stdio.h>

#pragma comment(lib, "Ws2_32.lib")

int main() {
    return 0;
}
```

Next Step: [Initializing Winsock](#)

## Related topics

[Getting Started With Winsock](#)

[About Servers and Clients](#)

# Initializing Winsock

3/5/2021 • 2 minutes to read • [Edit Online](#)

All processes (applications or DLLs) that call Winsock functions must initialize the use of the Windows Sockets DLL before making other Winsock functions calls. This also makes certain that Winsock is supported on the system.

## To initialize Winsock

1. Create a [WSADATA](#) object called wsaData.

```
WSADATA wsaData;
```

2. Call [WSAStartup](#) and return its value as an integer and check for errors.

```
int iResult;

// Initialize Winsock
iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
if (iResult != 0) {
    printf("WSAStartup failed: %d\n", iResult);
    return 1;
}
```

The [WSAStartup](#) function is called to initiate use of WS2\_32.dll.

The [WSADATA](#) structure contains information about the Windows Sockets implementation. The MAKEWORD(2,2) parameter of [WSAStartup](#) makes a request for version 2.2 of Winsock on the system, and sets the passed version as the highest version of Windows Sockets support that the caller can use.

Next Step for a Client: [Creating a Socket for the Client](#)

Next Step for a Server: [Creating a Socket for the Server](#)

## Related topics

[Getting Started With Winsock](#)

[Creating a Basic Winsock Application](#)

# Winsock Client Application

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following sections describe the remaining steps for creating a Winsock client application. What follows is the general model for creating a streaming TCP/IP client.

- [Creating a Socket for the Client](#)
- [Connecting to a Socket](#)
- [Sending and Receiving Data on the Client](#)
- [Disconnecting the Client](#)

## Complete Client Source Code

- [Complete Client Code](#)

## Related topics

[Getting Started With Winsock](#)

[Initializing Winsock](#)

# Creating a socket for the client

3/5/2021 • 2 minutes to read • [Edit Online](#)

After initialization, a **SOCKET** object must be instantiated for use by the client.

## To create a socket

1. Declare an **addrinfo** object that contains a **sockaddr** structure and initialize these values. For this application, the Internet address family is unspecified so that either an IPv6 or IPv4 address can be returned. The application requests the socket type to be a stream socket for the TCP protocol.

```
struct addrinfo *result = NULL,
    *ptr = NULL,
    hints;

ZeroMemory( &hints, sizeof(hints) );
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
```

2. Call the **getaddrinfo** function requesting the IP address for the server name passed on the command line. The TCP port on the server that the client will connect to is defined by **DEFAULT\_PORT** as 27015 in this sample. The **getaddrinfo** function returns its value as an integer that is checked for errors.

```
#define DEFAULT_PORT "27015"

// Resolve the server address and port
iResult = getaddrinfo(argv[1], DEFAULT_PORT, &hints, &result);
if (iResult != 0) {
    printf("getaddrinfo failed: %d\n", iResult);
    WSACleanup();
    return 1;
}
```

3. Create a **SOCKET** object called **ConnectSocket**.

```
SOCKET ConnectSocket = INVALID_SOCKET;
```

4. Call the **socket** function and return its value to the **ConnectSocket** variable. For this application, use the first IP address returned by the call to **getaddrinfo** that matched the address family, socket type, and protocol specified in the *hints* parameter. In this example, a TCP stream socket was specified with a socket type of **SOCK\_STREAM** and a protocol of **IPPROTO\_TCP**. The address family was left unspecified (**AF\_UNSPEC**), so the returned IP address could be either an IPv6 or IPv4 address for the server.

If the client application wants to connect using only IPv6 or IPv4, then the address family needs to be set to **AF\_INET6** for IPv6 or **AF\_INET** for IPv4 in the *hints* parameter.

```
// Attempt to connect to the first address returned by
// the call to getaddrinfo
ptr=result;

// Create a SOCKET for connecting to server
ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype,
    ptr->ai_protocol);
```

## 5. Check for errors to ensure that the socket is a valid socket.

```
if (ConnectSocket == INVALID_SOCKET) {
    printf("Error at socket(): %ld\n", WSAGetLastError());
    freeaddrinfo(result);
    WSACleanup();
    return 1;
}
```

The parameters passed to the [socket](#) function can be changed for different implementations.

Error detection is a key part of successful networking code. If the [socket](#) call fails, it returns INVALID\_SOCKET. The if statement in the previous code is used to catch any errors that may have occurred while creating the socket. [WSAGetLastError](#) returns an error number associated with the last error that occurred.

### NOTE

More extensive error checking may be necessary depending on the application.

For example, setting *hints.ai\_family* to AF\_UNSPEC can cause the connect call to fail. If that happens, then use a specific IPv4 (AF\_INET) or IPv6 (AF\_INET6) value instead.

[WSACleanup](#) is used to terminate the use of the WS2\_32 DLL.

Next Step: [Connecting to a Socket](#)

## Related topics

[Getting Started With Winsock](#)

[Initializing Winsock](#)

[Winsock Client Application](#)

# Connecting to a Socket

3/5/2021 • 2 minutes to read • [Edit Online](#)

For a client to communicate on a network, it must connect to a server.

## To connect to a socket

Call the [connect](#) function, passing the created socket and the [sockaddr](#) structure as parameters. Check for general errors.

```
// Connect to server.  
iResult = connect( ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);  
if (iResult == SOCKET_ERROR) {  
    closesocket(ConnectSocket);  
    ConnectSocket = INVALID_SOCKET;  
}  
  
// Should really try the next address returned by getaddrinfo  
// if the connect call failed  
// But for this simple example we just free the resources  
// returned by getaddrinfo and print an error message  
  
freeaddrinfo(result);  
  
if (ConnectSocket == INVALID_SOCKET) {  
    printf("Unable to connect to server!\n");  
    WSACleanup();  
    return 1;  
}
```

The [getaddrinfo](#) function is used to determine the values in the [sockaddr](#) structure. In this example, the first IP address returned by the [getaddrinfo](#) function is used to specify the [sockaddr](#) structure passed to the [connect](#). If the [connect](#) call fails to the first IP address, then try the next [addrinfo](#) structure in the linked list returned from the [getaddrinfo](#) function.

The information specified in the [sockaddr](#) structure includes:

- the IP address of the server that the client will try to connect to.
- the port number on the server that the client will connect to. This port was specified as port 27015 when the client called the [getaddrinfo](#) function.

Next Step: [Sending and Receiving Data on the Client](#)

## Related topics

[Getting Started With Winsock](#)

[Winsock Client Application](#)

[Creating a Socket for the Client](#)

# Sending and Receiving Data on the Client

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following code demonstrates the **send** and **recv** functions used by the client once a connection is established.

## Client

```
#define DEFAULT_BUFLEN 512

int recvbuf[DEFAULT_BUFLEN];
const char *sendbuf = "this is a test";
char recvbuf[DEFAULT_BUFLEN];

int iResult;

// Send an initial buffer
iResult = send(ConnectSocket, sendbuf, (int) strlen(sendbuf), 0);
if (iResult == SOCKET_ERROR) {
    printf("send failed: %d\n", WSAGetLastError());
    closesocket(ConnectSocket);
    WSACleanup();
    return 1;
}

printf("Bytes Sent: %ld\n", iResult);

// shutdown the connection for sending since no more data will be sent
// the client can still use the ConnectSocket for receiving data
iResult = shutdown(ConnectSocket, SD_SEND);
if (iResult == SOCKET_ERROR) {
    printf("shutdown failed: %d\n", WSAGetLastError());
    closesocket(ConnectSocket);
    WSACleanup();
    return 1;
}

// Receive data until the server closes the connection
do {
    iResult = recv(ConnectSocket, recvbuf, recvbuf[0], 0);
    if (iResult > 0)
        printf("Bytes received: %d\n", iResult);
    else if (iResult == 0)
        printf("Connection closed\n");
    else
        printf("recv failed: %d\n", WSAGetLastError());
} while (iResult > 0);
```

The **send** and **recv** functions both return an integer value of the number of bytes sent or received, respectively, or an error. Each function also takes the same parameters: the active socket, a **char** buffer, the number of bytes to send or receive, and any flags to use.

Next Step: [Disconnecting the Client](#)

## Related topics

[Getting Started With Winsock](#)

Winsock Client Application

Connecting to a Socket

# Disconnecting the Client

3/5/2021 • 2 minutes to read • [Edit Online](#)

Once the client is completed sending and receiving data, the client disconnects from the server and shutdowns the socket.

## To disconnect and shutdown a socket

- When the client is done sending data to the server, the [shutdown](#) function can be called specifying SD\_SEND to shutdown the sending side of the socket. This allows the server to release some of the resources for this socket. The client application can still receive data on the socket.

```
// shutdown the send half of the connection since no more data will be sent
iResult = shutdown(ConnectSocket, SD_SEND);
if (iResult == SOCKET_ERROR) {
    printf("shutdown failed: %d\n", WSAGetLastError());
    closesocket(ConnectSocket);
    WSACleanup();
    return 1;
}
```

- When the client application is done receiving data, the [closesocket](#) function is called to close the socket.

When the client application is completed using the Windows Sockets DLL, the [WSACleanup](#) function is called to release resources.

```
// cleanup
closesocket(ConnectSocket);
WSACleanup();

return 0;
```

## Complete Client Source Code

- [Complete Client Code](#)

## Related topics

[Getting Started With Winsock](#)

[Winsock Client Application](#)

[Sending and Receiving Data on the Client](#)

# Winsock Server Application

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following sections describe the remaining steps for creating a Winsock server application. What follows is the general model for creating a streaming TCP/IP server.

- [Creating a Socket for the Server](#)
- [Binding a Socket](#)
- [Listening on a Socket](#)
- [Accepting a Connection](#)
- [Receiving and Sending Data on the Server](#)
- [Disconnecting the Server](#)

## Complete Server Source Code

- [Complete Server Code](#)

## Related topics

[Getting Started With Winsock](#)

[Initializing Winsock](#)

# Creating a Socket for the Server

3/5/2021 • 2 minutes to read • [Edit Online](#)

After initialization, a **SOCKET** object must be instantiated for use by the server.

## To create a socket for the server

1. The **getaddrinfo** function is used to determine the values in the **sockaddr** structure:

- **AF\_INET** is used to specify the IPv4 address family.
- **SOCK\_STREAM** is used to specify a stream socket.
- **IPPROTO\_TCP** is used to specify the TCP protocol .
- **AI\_PASSIVE** flag indicates the caller intends to use the returned socket address structure in a call to the **bind** function. When the **AI\_PASSIVE** flag is set and *nodename* parameter to the **getaddrinfo** function is a **NULL** pointer, the IP address portion of the socket address structure is set to **INADDR\_ANY** for IPv4 addresses or **IN6ADDR\_ANY\_INIT** for IPv6 addresses.
- 27015 is the port number associated with the server that the client will connect to.

The **addrinfo** structure is used by the **getaddrinfo** function.

```
#define DEFAULT_PORT "27015"

struct addrinfo *result = NULL, *ptr = NULL, hints;

ZeroMemory(&hints, sizeof (hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags = AI_PASSIVE;

// Resolve the local address and port to be used by the server
iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
if (iResult != 0) {
    printf("getaddrinfo failed: %d\n", iResult);
    WSACleanup();
    return 1;
}
```

2. Create a **SOCKET** object called ListenSocket for the server to listen for client connections.

```
SOCKET ListenSocket = INVALID_SOCKET;
```

3. Call the **socket** function and return its value to the ListenSocket variable. For this server application, use the first IP address returned by the call to **getaddrinfo** that matched the address family, socket type, and protocol specified in the *hints* parameter. In this example, a TCP stream socket for IPv4 was requested with an address family of IPv4, a socket type of **SOCK\_STREAM** and a protocol of **IPPROTO\_TCP**. So an IPv4 address is requested for the ListenSocket.

If the server application wants to listen on IPv6, then the address family needs to be set to **AF\_INET6** in the *hints* parameter. If a server wants to listen on both IPv6 and IPv4, two listen sockets must be created, one for IPv6 and one for IPv4. These two sockets must be handled separately by the application.

Windows Vista and later offer the ability to create a single IPv6 socket that is put in dual stack mode to listen on both IPv6 and IPv4. For more information on this feature, see [Dual-Stack Sockets](#).

```
// Create a SOCKET for the server to listen for client connections  
  
ListenSocket = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
```

4. Check for errors to ensure that the socket is a valid socket.

```
if (ListenSocket == INVALID_SOCKET) {  
    printf("Error at socket(): %ld\n", WSAGetLastError());  
    freeaddrinfo(result);  
    WSACleanup();  
    return 1;  
}
```

Next Step: [Binding a Socket](#)

## Related topics

[Getting Started With Winsock](#)

[Initializing Winsock](#)

[Winsock Server Application](#)

# Binding a Socket

3/5/2021 • 2 minutes to read • [Edit Online](#)

For a server to accept client connections, it must be bound to a network address within the system. The following code demonstrates how to bind a socket that has already been created to an IP address and port. Client applications use the IP address and port to connect to the host network.

## To bind a socket

The **sockaddr** structure holds information regarding the address family, IP address, and port number.

Call the **bind** function, passing the created **socket** and **sockaddr** structure returned from the **getaddrinfo** function as parameters. Check for general errors.

```
// Setup the TCP listening socket
iResult = bind( ListenSocket, result->ai_addr, (int)result->ai_addrlen);
if (iResult == SOCKET_ERROR) {
    printf("bind failed with error: %d\n", WSAGetLastError());
    freeaddrinfo(result);
    closesocket(ListenSocket);
    WSACleanup();
    return 1;
}
```

Once the **bind** function is called, the address information returned by the **getaddrinfo** function is no longer needed. The **freeaddrinfo** function is called to free the memory allocated by the **getaddrinfo** function for this address information.

```
freeaddrinfo(result);
```

Next Step: [Listening on a Socket](#)

## Related topics

[Getting Started With Winsock](#)

[Winsock Server Application](#)

[Creating a Socket for the Server](#)

# Listening on a Socket

3/5/2021 • 2 minutes to read • [Edit Online](#)

After the socket is bound to an IP address and port on the system, the server must then listen on that IP address and port for incoming connection requests.

## To listen on a socket

Call the [listen](#) function, passing as parameters the created socket and a value for the *backlog*, maximum length of the queue of pending connections to accept. In this example, the *backlog* parameter was set to **SOMAXCONN**. This value is a special constant that instructs the Winsock provider for this socket to allow a maximum reasonable number of pending connections in the queue. Check the return value for general errors.

```
if ( listen( ListenSocket, SOMAXCONN ) == SOCKET_ERROR ) {
    printf( "Listen failed with error: %ld\n", WSAGetLastError() );
    closesocket(ListenSocket);
    WSACleanup();
    return 1;
}
```

Next Step: [Accepting a Connection](#)

## Related topics

[Getting Started With Winsock](#)

[Winsock Server Application](#)

[Binding a Socket](#)

# Accepting a Connection (Windows Sockets 2)

3/5/2021 • 2 minutes to read • [Edit Online](#)

Once the socket is listening for a connection, the program must handle connection requests on that socket.

## To accept a connection on a socket

1. Create a temporary **SOCKET** object called ClientSocket for accepting connections from clients.

```
SOCKET ClientSocket;
```

2. Normally a server application would be designed to listen for connections from multiple clients. For high-performance servers, multiple threads are commonly used to handle the multiple client connections.

There are several different programming techniques using Winsock that can be used to listen for multiple client connections. One programming technique is to create a continuous loop that checks for connection requests using the [listen](#) function (see [Listening on a Socket](#)). If a connection request occurs, the application calls the [accept](#), [AcceptEx](#), or [WSAAccept](#) function and passes the work to another thread to handle the request. Several other programming techniques are possible.

Note that this basic example is very simple and does not use multiple threads. The example also just listens for and accepts only a single connection.

```
ClientSocket = INVALID_SOCKET;

// Accept a client socket
ClientSocket = accept(ListenSocket, NULL, NULL);
if (ClientSocket == INVALID_SOCKET) {
    printf("accept failed: %d\n", WSAGetLastError());
    closesocket(ListenSocket);
    WSACleanup();
    return 1;
}
```

3. When the client connection has been accepted, a server application would normally pass the accepted client socket (the ClientSocket variable in the above sample code) to a worker thread or an I/O completion port and continue accepting additional connections. In this basic example, the server continues to the next step.

There are a number of other programming techniques that can be used to listen for and accept multiple connections. These include using the [select](#) or [WSAPoll](#) functions. Examples of some of these various programming techniques are illustrated in the [Advanced Winsock Samples](#) included with the Microsoft Windows Software Development Kit (SDK).

#### **NOTE**

On Unix systems, a common programming technique for servers was for an application to listen for connections. When a connection was accepted, the parent process would call the **fork** function to create a new child process to handle the client connection, inheriting the socket from the parent. This programming technique is not supported on Windows, since the **fork** function is not supported. This technique is also not usually suitable for high-performance servers, since the resources needed to create a new process are much greater than those needed for a thread.

Next Step: [Receiving and Sending Data on the Server](#)

## Related topics

[Getting Started With Winsock](#)

[Winsock Server Application](#)

[Listening on a Socket](#)

# Receiving and Sending Data on the Server

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following code demonstrates the `recv` and `send` functions used by the server.

## To receive and send data on a socket

```
#define DEFAULT_BUFLEN 512

char recvbuf[DEFAULT_BUFLEN];
int iResult, iSendResult;
int recvbuflen = DEFAULT_BUFLEN;

// Receive until the peer shuts down the connection
do {

    iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);
    if (iResult > 0) {
        printf("Bytes received: %d\n", iResult);

        // Echo the buffer back to the sender
        iSendResult = send(ClientSocket, recvbuf, iResult, 0);
        if (iSendResult == SOCKET_ERROR) {
            printf("send failed: %d\n", WSAGetLastError());
            closesocket(ClientSocket);
            WSACleanup();
            return 1;
        }
        printf("Bytes sent: %d\n", iSendResult);
    } else if (iResult == 0)
        printf("Connection closing...\n");
    else {
        printf("recv failed: %d\n", WSAGetLastError());
        closesocket(ClientSocket);
        WSACleanup();
        return 1;
    }

} while (iResult > 0);
```

The `send` and `recv` functions both return an integer value of the number of bytes sent or received, respectively, or an error. Each function also takes the same parameters: the active socket, a `char` buffer, the number of bytes to send or receive, and any flags to use.

Next Step: [Disconnecting the Server](#)

## Related topics

[Getting Started With Winsock](#)

[Winsock Server Application](#)

[Accepting a Connection](#)

# Disconnecting the Server

3/5/2021 • 2 minutes to read • [Edit Online](#)

Once the server is completed receiving data from the client and sending data back to the client, the server disconnects from the client and shutdowns the socket.

## To disconnect and shutdown a socket

- When the server is done sending data to the client, the [shutdown](#) function can be called specifying SD\_SEND to shutdown the sending side of the socket. This allows the client to release some of the resources for this socket. The server application can still receive data on the socket.

```
// shutdown the send half of the connection since no more data will be sent
iResult = shutdown(ClientSocket, SD_SEND);
if (iResult == SOCKET_ERROR) {
    printf("shutdown failed: %d\n", WSAGetLastError());
    closesocket(ClientSocket);
    WSACleanup();
    return 1;
}
```

- When the client application is done receiving data, the [closesocket](#) function is called to close the socket.

When the client application is completed using the Windows Sockets DLL, the [WSACleanup](#) function is called to release resources.

```
// cleanup
closesocket(ClientSocket);
WSACleanup();

return 0;
```

## Complete Server Source Code

- [Complete Server Code](#)

## Related topics

[Getting Started With Winsock](#)

[Winsock Server Application](#)

[Receiving and Sending Data on the Server](#)

[Running the Winsock Client and Server Code Sample](#)

# Running the Winsock Client and Server Code Sample

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section contains the complete source code for the TCP/IP Client and Server applications:

- [Complete Winsock Client Code](#)
- [Complete Winsock Server Code](#)

The server application should be started before the client application is started.

To execute the server, compile the complete server source code and run the executable file. The server application listens on TCP port 27015 for a client to connect. Once a client connects, the server receives data from the client and echoes (sends) the data received back to the client. When the client shuts down the connection, the server shuts down the client socket, closes the socket, and exits.

To execute the client, compile the complete client source code and run the executable file. The client application requires that name of the computer or IP address of the computer where the server application is running is passed as a command-line parameter when the client is executed. If the client and server are executed on the sample computer, the client can be started as follows:

**client localhost**

The client tries to connect to the server on TCP port 27015. Once the client connects, the client sends data to the server and receives any data send back from the server. The client then closes the socket and exits.

## Related topics

[Getting Started With Winsock](#)

# Complete Winsock Client Code

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following is the complete source code for the basic Winsock TCP/IP Client Application.

## Winsock Client Source Code

```
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>

// Need to link with Ws2_32.lib, Mswsock.lib, and Advapi32.lib
#pragma comment (lib, "Ws2_32.lib")
#pragma comment (lib, "Mswsock.lib")
#pragma comment (lib, "AdvApi32.lib")

#define DEFAULT_BUFLEN 512
#define DEFAULT_PORT "27015"

int __cdecl main(int argc, char **argv)
{
    WSADATA wsaData;
    SOCKET ConnectSocket = INVALID_SOCKET;
    struct addrinfo *result = NULL,
        *ptr = NULL,
        hints;
    const char *sendbuf = "this is a test";
    char recvbuf[DEFAULT_BUFLEN];
    int iResult;
    int recvbuflen = DEFAULT_BUFLEN;

    // Validate the parameters
    if (argc != 2) {
        printf("usage: %s server-name\n", argv[0]);
        return 1;
    }

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed with error: %d\n", iResult);
        return 1;
    }

    ZeroMemory( &hints, sizeof(hints) );
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    // Resolve the server address and port
    iResult = getaddrinfo(argv[1], DEFAULT_PORT, &hints, &result);
    if ( iResult != 0 ) {
        printf("getaddrinfo failed with error: %d\n", iResult);
        WSACleanup();
        return 1;
    }
```

```

}

// Attempt to connect to an address until one succeeds
for(ptr=result; ptr != NULL ;ptr=ptr->ai_next) {

    // Create a SOCKET for connecting to server
    ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype,
                           ptr->ai_protocol);
    if (ConnectSocket == INVALID_SOCKET) {
        printf("socket failed with error: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    // Connect to server.
    iResult = connect( ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
    if (iResult == SOCKET_ERROR) {
        closesocket(ConnectSocket);
        ConnectSocket = INVALID_SOCKET;
        continue;
    }
    break;
}

freeaddrinfo(result);

if (ConnectSocket == INVALID_SOCKET) {
    printf("Unable to connect to server!\n");
    WSACleanup();
    return 1;
}

// Send an initial buffer
iResult = send( ConnectSocket, sendbuf, (int)strlen(sendbuf), 0 );
if (iResult == SOCKET_ERROR) {
    printf("send failed with error: %d\n", WSAGetLastError());
    closesocket(ConnectSocket);
    WSACleanup();
    return 1;
}

printf("Bytes Sent: %ld\n", iResult);

// shutdown the connection since no more data will be sent
iResult = shutdown(ConnectSocket, SD_SEND);
if (iResult == SOCKET_ERROR) {
    printf("shutdown failed with error: %d\n", WSAGetLastError());
    closesocket(ConnectSocket);
    WSACleanup();
    return 1;
}

// Receive until the peer closes the connection
do {

    iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
    if ( iResult > 0 )
        printf("Bytes received: %d\n", iResult);
    else if ( iResult == 0 )
        printf("Connection closed\n");
    else
        printf("recv failed with error: %d\n", WSAGetLastError());

} while( iResult > 0 );

// cleanup
closesocket(ConnectSocket);
WSACleanup();

```

```
    return 0;  
}
```

## Related topics

[Getting Started With Winsock](#)

[Running the Winsock Client and Server Code Sample](#)

[Complete Winsock Server Code](#)

# Complete Winsock Server Code

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following is the complete source code for the basic Winsock TCP/IP Server application.

## Winsock Server Source Code

```
#undef UNICODE

#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdlib.h>
#include <stdio.h>

// Need to link with Ws2_32.lib
#pragma comment (lib, "Ws2_32.lib")
// #pragma comment (lib, "Mswsock.lib")

#define DEFAULT_BUflen 512
#define DEFAULT_PORT "27015"

int __cdecl main(void)
{
    WSADATA wsaData;
    int iResult;

    SOCKET ListenSocket = INVALID_SOCKET;
    SOCKET ClientSocket = INVALID_SOCKET;

    struct addrinfo *result = NULL;
    struct addrinfo hints;

    int iSendResult;
    char recvbuf[DEFAULT_BUflen];
    int recvbuflen = DEFAULT_BUflen;

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed with error: %d\n", iResult);
        return 1;
    }

    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = AI_PASSIVE;

    // Resolve the server address and port
    iResult = getaddrinfo(NULL, DEFAULT_PORT, &hints, &result);
    if ( iResult != 0 ) {
        printf("getaddrinfo failed with error: %d\n", iResult);
        WSACleanup();
        return 1;
    }

    // Create a SOCKET for connecting to server
    // ... (rest of the code)
```

```

ListenSocket = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
if (ListenSocket == INVALID_SOCKET) {
    printf("socket failed with error: %d\n", WSAGetLastError());
    freeaddrinfo(result);
    WSACleanup();
    return 1;
}

// Setup the TCP listening socket
iResult = bind( ListenSocket, result->ai_addr, (int)result->ai_addrlen);
if (iResult == SOCKET_ERROR) {
    printf("bind failed with error: %d\n", WSAGetLastError());
    freeaddrinfo(result);
    closesocket(ListenSocket);
    WSACleanup();
    return 1;
}

freeaddrinfo(result);

iResult = listen(ListenSocket, SOMAXCONN);
if (iResult == SOCKET_ERROR) {
    printf("listen failed with error: %d\n", WSAGetLastError());
    closesocket(ListenSocket);
    WSACleanup();
    return 1;
}

// Accept a client socket
ClientSocket = accept(ListenSocket, NULL, NULL);
if (ClientSocket == INVALID_SOCKET) {
    printf("accept failed with error: %d\n", WSAGetLastError());
    closesocket(ListenSocket);
    WSACleanup();
    return 1;
}

// No longer need server socket
closesocket(ListenSocket);

// Receive until the peer shuts down the connection
do {

    iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);
    if (iResult > 0) {
        printf("Bytes received: %d\n", iResult);

        // Echo the buffer back to the sender
        iSendResult = send( ClientSocket, recvbuf, iResult, 0 );
        if (iSendResult == SOCKET_ERROR) {
            printf("send failed with error: %d\n", WSAGetLastError());
            closesocket(ClientSocket);
            WSACleanup();
            return 1;
        }
        printf("Bytes sent: %d\n", iSendResult);
    }
    else if (iResult == 0)
        printf("Connection closing...\n");
    else {
        printf("recv failed with error: %d\n", WSAGetLastError());
        closesocket(ClientSocket);
        WSACleanup();
        return 1;
    }

} while (iResult > 0);

// shutdown the connection since we're done

```

```
iResult = shutdown(ClientSocket, SD_SEND);
if (iResult == SOCKET_ERROR) {
    printf("shutdown failed with error: %d\n", WSAGetLastError());
    closesocket(ClientSocket);
    WSACleanup();
    return 1;
}

// cleanup
closesocket(ClientSocket);
WSACleanup();

return 0;
}
```

## Related topics

[Getting Started With Winsock](#)

[Running the Winsock Client and Server Code Sample](#)

[Complete Winsock Client Code](#)

# Secure Winsock Programming

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following is a guide to secure Windows Sockets programming. It is designed to provide an understanding of Winsock security and the options available to the secure network application developer.

- [Using SO\\_REUSEADDR and SO\\_EXCLUSIVEADDRUSE](#)
- [Winsock Secure Socket Extensions](#)

Communications using sockets can also be encrypted using the SSL/TLS standards using Secure Channel, also known as Schannel technology. Schannel is a security support provider (SSP) that contains a set of security protocols that provide identity authentication and secure, private communication through encryption. Schannel is primarily used for Internet applications that require secure Hypertext Transfer Protocol (HTTP) communications. For more information, please see [Secure Channel](#).

## Related topics

[Secure Channel](#)

# Using SO\_REUSEADDR and SO\_EXCLUSIVEADDRUSE

3/5/2021 • 12 minutes to read • [Edit Online](#)

Developing a secure high-level network infrastructure is a priority for most network application developers. However, socket security is often overlooked despite being very critical when considering a fully secure solution. Socket security, specifically, deals with processes that bind to the same port previously bound by another application process. In the past, it was possible for a network application to "hijack" the port of another application, which could easily lead to a "denial of service" attack or data theft.

In general, socket security applies to server-side processes. More specifically, socket security applies to any network application that accepts connections and receives IP datagram traffic. These applications typically bind to a well-known port and are common targets for malicious network code.

Client applications are less likely to be the targets of such attacks — not because they are less susceptible, but because most clients bind to "ephemeral" local ports rather than static "service" ports. Client network applications should always bind to ephemeral ports (by specifying port 0 in the **SOCKADDR** structure pointed to by the *name* parameter when calling the **bind** function) unless there is a compelling architectural reason not to. The ephemeral local ports consist of ports greater than port 49151. Most server applications for dedicated services bind to a well-known reserved port that is less than or equal to port 49151. So for most applications, there is not usually a conflict for bind requests between client and server applications.

This section describes the default level of security on various Microsoft Windows platforms and how the specific socket options **SO\_REUSEADDR** and **SO\_EXCLUSIVEADDRUSE** impact and affect network application security. An additional feature called enhanced socket security is available on Windows Server 2003 and later. The availability of these socket options and enhanced socket security varies across versions of Microsoft operating systems, as shown in the table below.

PLATFORM	SO_REUSEADDR	SO_EXCLUSIVEADDRUSE	ENHANCED SOCKET SECURITY
Windows 95	Available	Not Available	Not Available
Windows 98	Available	Not Available	Not Available
Windows Me	Available	Not Available	Not Available
Windows NT 4.0	Available	Available in Service Pack 4 and later	Not Available
Windows 2000	Available	Available	Not Available
Windows XP	Available	Available	Not Available
Windows Server 2003	Available	Available	Available
Windows Vista	Available	Available	Available
Windows Server 2008	Available	Available	Available

PLATFORM	SO_REUSEADDR	SO_EXCLUSIVEADDRUSE	ENHANCED SOCKET SECURITY
Windows 7 and newer	Available	Available	Available

## Using SO\_REUSEADDR

The **SO\_REUSEADDR** socket option allows a socket to forcibly bind to a port in use by another socket. The second socket calls [setsockopt](#) with the *optname* parameter set to **SO\_REUSEADDR** and the *optval*/parameter set to a boolean value of **TRUE** before calling **bind** on the same port as the original socket. Once the second socket has successfully bound, the behavior for all sockets bound to that port is indeterminate. For example, if all of the sockets on the same port provide TCP service, any incoming TCP connection requests over the port cannot be guaranteed to be handled by the correct socket — the behavior is non-deterministic. A malicious program can use **SO\_REUSEADDR** to forcibly bind sockets already in use for standard network protocol services in order to deny access to those service. No special privileges are required to use this option.

If a client application binds to a port before a server application is able to bind to the same port, then problems may result. If the server application forcibly binds using the **SO\_REUSEADDR** socket option to the same port, then the behavior for all sockets bound to that port is indeterminate.

The exception to this non-deterministic behavior is multicast sockets. If two sockets are bound to the same interface and port and are members of the same multicast group, data will be delivered to both sockets, rather than an arbitrarily chosen one.

## Using SO\_EXCLUSIVEADDRUSE

Before the **SO\_EXCLUSIVEADDRUSE** socket option was introduced, there was very little a network application developer could do to prevent a malicious program from binding to the port on which the network application had its own sockets bound. In order to address this security issue, Windows Sockets introduced the **SO\_EXCLUSIVEADDRUSE** socket option, which became available on Windows NT 4.0 with Service Pack 4 (SP4) and later.

The **SO\_EXCLUSIVEADDRUSE** socket option can only be used by members of the Administrators security group on Windows XP and earlier. The reasons why this requirement was changed on Windows Server 2003 and later are discussed later in the article.

The **SO\_EXCLUSIVEADDRUSE** option is set by calling the [setsockopt](#) function with the *optname* parameter set to **SO\_EXCLUSIVEADDRUSE** and the *optval*/parameter set to a boolean value of **TRUE** before the socket is bound. After the option is set, the behavior of subsequent **bind** calls differs depending on the network address specified in each **bind** call.

The table below describes the behavior that occurs in Windows XP and earlier when a second socket attempts to bind to an address previously bound to by a first socket using specific socket options.

### NOTE

In the table below, "wildcard" denotes the wildcard address for the given protocol (such as "0.0.0.0" for IPv4 and ":" for IPv6). "Specific" denotes a specific IP address assigned an interface. The table cells indicate whether or not the bind is successful ("Success") or an error is returned ("INUSE" for the [WSAEADDRINUSE](#) error; "ACCESS" for the [WSAEACCES](#) error).

First <b>bind</b> call	Second <b>bind</b> call

		Default		SO_REUSEADDR		SO_EXCLUSIVEADDRUSE	
		Wildcard	Specific	Wildcard	Specific	Wildcard	Specific
Default	Wildcard	INUSE	INUSE	Success	Success	INUSE	INUSE
	Specific	INUSE	INUSE	Success	Success	INUSE	INUSE
SO_REUSEADDR	Wildcard	INUSE	INUSE	Success	Success	INUSE	INUSE
	Specific	INUSE	INUSE	Success	Success	INUSE	INUSE
SO_EXCLUSIVEADDRUSE	Wildcard	INUSE	INUSE	ACCESS	ACCESS	INUSE	INUSE
	Specific	INUSE	INUSE	ACCESS	ACCESS	INUSE	INUSE

When two sockets are bound to the same port number but on different explicit interfaces, there is no conflict. For example, in the case where a computer has two IP interfaces, 10.0.0.1 and 10.99.99.99, if the first call to **bind** is on 10.0.0.1 with the port set to 5150 and **SO\_EXCLUSIVEADDRUSE** specified, then a second call to **bind** on 10.99.99.99 with the port also set to 5150 and no options specified will succeed. However, if the first socket is bound to the wildcard address and port 5150, then any subsequent bind call to port 5150 with **SO\_EXCLUSIVEADDRUSE** set will fail with either **WSAEADDRINUSE** or **WSAEACCES** returned by the **bind** operation.

In the case where the first call to **bind** sets either **SO\_REUSEADDR** or no socket options at all, the second **bind** call will "hijack" the port and the application will be unable to determine which of the two sockets received specific packets sent to the "shared" port.

A typical application that calls the **bind** function does not allocate the bound socket for exclusive use, unless the **SO\_EXCLUSIVEADDRUSE** socket option is called on the socket before the call to the **bind** function. If a client application binds to an ephemeral port or a specific port before a server application binds to the same port, then problems can result. The server application can forcibly bind to the same port by using the **SO\_REUSEADDR** socket option on the socket before calling the **bind** function, but the behavior for all sockets bound to that port is then indeterminate. If the server application tries to use the **SO\_EXCLUSIVEADDRUSE** socket option for exclusive use of the port, the request will fail.

Conversely, a socket with the **SO\_EXCLUSIVEADDRUSE** set cannot necessarily be reused immediately after socket closure. For example, if a listening socket with **SO\_EXCLUSIVEADDRUSE** set accepts a connection and is then subsequently closed, another socket (also with **SO\_EXCLUSIVEADDRUSE**) cannot bind to the same port as the first socket until the original connection becomes inactive.

This issue can become complicated because the underlying transport protocol may not terminate the connection even though the socket has been closed. Even after the socket has been closed by the application, the system must transmit any buffered data, send a graceful disconnect message to the peer, and wait for a corresponding graceful disconnect message from the peer. It is possible that the underlying transport protocol might never release the connection; for example, the peer participating in the original connection might advertise a zero-size window, or some other form of "attack" configuration. In such a case, the client connection remains in an active state despite the request to close it, since unacknowledged data remains in the buffer.

To avoid this situation, network applications should ensure a graceful shutdown by calling **shutdown** with the **SD\_SEND** flag set, and then wait in a **recv** loop until zero bytes are returned over the connection. This guarantees that all data is received by the peer and likewise confirms with the peer that it has received all of the transmitted data, as well as avoiding the aforementioned port reuse issue.

The SO\_LINGER socket option may be set on a socket to prevent the port from transitioning to an "active" wait state; however, this is discouraged as it can lead to desired effects, such as reset connections. For example, if data is received by the peer but remains unacknowledged by it, and the local computer closes the socket with SO\_LINGER set on it, the connection between the two computers is reset and the unacknowledged data discarded by the peer. Picking a suitable time to linger is difficult as a smaller timeout value often results in suddenly aborted connections, whereas larger timeout values leave the system vulnerable to denial-of-service attacks (by establishing many connections and potentially stalling/blocking application threads). Closing a socket that has a nonzero linger timeout value may also cause the [closesocket](#) call to block.

## Enhanced Socket Security

Enhanced socket security was added with the release of Windows Server 2003. In previous Microsoft server operating system releases, the default socket security easily allowed processes to hijack ports from unsuspecting applications. In Windows Server 2003, sockets are not in a sharable state by default. Therefore, if an application wants to allow other processes to reuse a port on which a socket is already bound, it must specifically enable it. If that is the case, the first socket to call [bind](#) on the port must have SO\_REUSEADDR set on the socket. The only exception to this case occurs when the second [bind](#) call is performed by the same user account that made the original call to [bind](#). This exception exists solely to provide backward compatibility.

The table below describes the behavior that occurs in Windows Server 2003 and later operating systems when a second socket attempts to bind to an address previously bound to by a first socket using specific socket options.

### NOTE

In the table below, "wildcard" denotes the wildcard address for the given protocol (such as "0.0.0.0" for IPv4 and ":" for IPv6). "Specific" denotes a specific IP address assigned an interface. The table cells indicate whether or not the bind is successful ("Success") or the error returned ("INUSE" for the [WSAEADDRINUSE](#) error; "ACCESS" for the [WSAEACCES](#) error).

Also note that in this specific table, both [bind](#) calls are made under the same user account.

First <a href="#">bind</a> call		Second <a href="#">bind</a> call					
		Default		SO_REUSEADDR		SO_EXCLUSIVEADDRUSE	
		Wildcard	Specific	Wildcard	Specific	Wildcard	Specific
Default	Wildcard	INUSE	Success	ACCESS	Success	INUSE	Success
	Specific	Success	INUSE	Success	ACCESS	INUSE	INUSE
SO_REUSEADDR	Wildcard	INUSE	Success	Success	Success	INUSE	Success
	Specific	Success	INUSE	Success	Success	INUSE	INUSE
SO_EXCLUSIVEADDRUSE	Wildcard	INUSE	ACCESS	ACCESS	ACCESS	INUSE	ACCESS
	Specific	Success	INUSE	Success	ACCESS	INUSE	INUSE

A few entries in the table above merit explanation.

For example, if the first caller sets SO\_EXCLUSIVEADDRUSE on a specific address, and second caller attempts to call [bind](#) with a wildcard address on the same port, the second [bind](#) call will succeed. In this particular case, the second caller is bound to all interfaces except the specific address to which the first caller is bound. Note that

the reverse of this case is not true: if the first caller sets **SO\_EXCLUSIVEADDRUSE** and calls **bind** with the wildcard flag, the second caller is not able to call **bind** with the same port.

The socket binding behavior changes when the socket bind calls are made under different user accounts. The table below specifies the behavior that occurs in Windows Server 2003 and later operating systems when a second socket attempts to bind to an address previously bound to by a first socket using specific socket options and a different user account.

First <b>bind</b> call		Second <b>bind</b> call					
		Default		SO_REUSEADDR		SO_EXCLUSIVEADDRUSE	
		Wildcard	Specific	Wildcard	Specific	Wildcard	Specific
Default	Wildcard	INUSE	ACCESS	ACCESS	ACCESS	INUSE	ACCESS
	Specific	Success	INUSE	Success	ACCESS	INUSE	INUSE
SO_REUSEADDR	Wildcard	INUSE	ACCESS	Success	Success	INUSE	ACCESS
	Specific	Success	INUSE	Success	Success	INUSE	INUSE
SO_EXCLUSIVEADDRUSE	Wildcard	INUSE	ACCESS	ACCESS	ACCESS	INUSE	ACCESS
	Specific	Success	INUSE	Success	ACCESS	INUSE	INUSE

Note that the default behavior is different when the **bind** calls are made under different user accounts. If the first caller does not set any options on the socket and binds to the wildcard address, then the second caller cannot set the **SO\_REUSEADDR** option and successfully bind to the same port. The default behavior with no options set returns an error, as well.

On Windows Vista and later, a dual stack socket can be created which operates over both IPv6 and IPv4. When a dual stack socket is bound to the wildcard address, the given port is reserved on both the IPv4 and IPv6 networking stacks and the checks associated with **SO\_REUSEADDR** and **SO\_EXCLUSIVEADDRUSE** (if set) are made. These checks must succeed on both networking stacks. For example, if a dual stack TCP socket sets **SO\_EXCLUSIVEADDRUSE** and then tries to bind to port 5000, then no other TCP socket can be previously bound to port 5000 (either wildcard or specific). In this case if an IPv4 TCP socket was previously bound to the loopback address on port 5000, the **bind** call for the dual stack socket would fail with [WSAEACCES](#).

## Application Strategies

When developing network application that operate at the socket layer, it is important to consider the type of socket security necessary. Client applications — applications that connect or send data to a service — rarely require any additional steps since they bind to a random local (ephemeral) port. If the client does require a specific local port binding in order to function correctly, then you must consider socket security.

The **SO\_REUSEADDR** option has very few uses in normal applications aside from multicast sockets where data is delivered to all of the sockets bound on the same port. Otherwise, any application that sets this socket option should be redesigned to remove the dependency since it is eminently vulnerable to "socket hijacking". As long as **SO\_REUSEADDR** socket option can be used to potentially hijack a port in a server application, the application must be considered to be not secure.

All server applications must set **SO\_EXCLUSIVEADDRUSE** for a strong level of socket security. Not only does it prevent malicious software from hijacking the port, but it also indicates whether or not another application is

bound to the requested port. For example, a call to [bind](#) on the wildcard address by a process with the `SO_EXCLUSIVEADDRUSE` socket option set will fail if another process is currently bound to the same port on a specific interface.

Lastly, even though socket security has been improved in Windows Server 2003, an application should always set the `SO_EXCLUSIVEADDRUSE` socket option to ensure that it binds to all the specific interfaces the process has requested. The socket security in Windows Server 2003 adds an increased level of security for legacy applications, but application developers must still design their products with all aspects of security in mind.

# Winsock Secure Socket Extensions

3/5/2021 • 3 minutes to read • [Edit Online](#)

The secure socket extensions to Winsock allow a socket application to control the security of their traffic over a network. These extensions allow an application to provide security policy and requirements for their network traffic, and query the security settings applied. For example, an application can use these extensions to query the peer security token that can be used to perform application level access checks.

The secure socket extensions are intended to integrate the services provided by IPsec and other security protocols with the Winsock framework. Prior to Windows Vista, on Windows Server 2003 and Windows XP, IPsec has been configured by an administrator via local and domain policies. On Windows Vista, the secure socket extensions instead allow applications to entirely or partially configure and control the security of their network traffic at the socket level.

Applications can already secure network traffic by using public APIs, such as IPsec management, Windows Filtering Platform and Security Support Provider Interface (SSPI). However, using these APIs may make the application more difficult to develop, and may make it more difficult to configure and deploy. The Winsock secure socket extensions have been designed to simplify the development of network applications that require secure network traffic by letting Winsock handle most of the complexity.

These secure socket extensions are available on Windows Vista and later.

## Secure Socket Functions

The secure socket extension functions are as follows:

- [WSADeleteSocketPeerTargetName](#)
- [WSAImpersonateSocketPeer](#)
- [WSAQuerySocketSecurity](#)
- [WSARevertImpersonation](#)
- [WSASetSocketPeerTargetName](#)
- [WSASetSocketSecurity](#)

### NOTE

The secure socket functions currently support only the IPsec protocol and are available on Windows Vista and later.

The structures and enumerations used by the secure socket functions are as follows:

- [SOCKET\\_PEER\\_TARGET\\_NAME](#)
- [SOCKET\\_SECURITY\\_PROTOCOL](#)
- [SOCKET\\_SECURITY\\_QUERY\\_INFO](#)
- [SOCKET\\_SECURITY\\_QUERY\\_TEMPLATE](#)
- [SOCKET\\_SECURITY\\_SETTINGS](#)
- [SOCKET\\_SECURITY\\_SETTINGS\\_IPSEC](#)

The secure socket functions are simple to use for normal applications and are flexible enough for applications that need a high degree of control over their security. These functions make it possible to keep the underlying

security mechanism hidden from the application. An application can specify generic security requirements and let the administrator control the security protocol that is used to support the requirements. While it is possible to extend these functions to add other security protocols, currently only IPsec integrates with the secure socket functions.

The [WSASetSocketSecurity](#) function allows an application to enable security and apply security settings before a connection is established.

The [WSASetSocketPeerTargetName](#) function allows an application to specify the target name corresponding to a peer entity. The selected security protocol will use this information when authenticating the peer. This feature addresses concerns about trusted man-in-the-middle attacks.

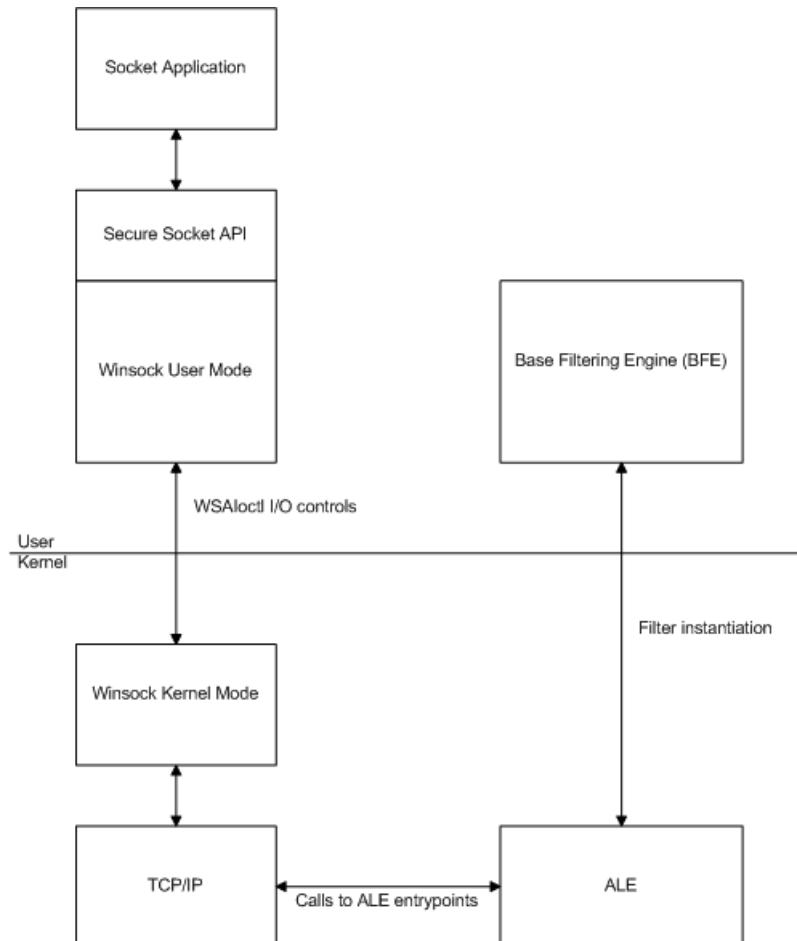
The [WSADeleteSocketPeerTargetName](#) function is used to delete a previously specified peer name for a socket.

After a connection is established, the [WSAQuerySocketSecurity](#) function allows an application to query the security properties of the connection, which can include the peer access or computer access token.

After a connection is established, the [WSAImpersonateSocketPeer](#) function allows an application to impersonate the security principal corresponding to a socket peer in order to perform application-level authorization.

The [WSARevertImpersonation](#) allows an application to terminate the impersonation of a socket peer.

## Secure Socket Architecture



- An application calls the secure socket functions to set or query security settings for a socket.
- The secure socket functions are a set of type-safe extension functions that wrap calls to the [WSAIoctl](#) function using newly-defined values for the *dwIoControlCode* parameter available on Windows Vista and later. These IOCTLs are handled by the network stack.

- The network stack will direct the call to [Application Layer Enforcement \(ALE\)](#) along with the endpoint handle. For the [WSADeleteSocketPeerTargetName](#), [WSASetSocketPeerTargetName](#), and [WSASetSocketSecurity](#) functions, ALE will configure the application's settings on the local endpoint. For the [WSAQuerySocketSecurity](#) function, ALE will read the requested information from applicable local and remote endpoints.
- Based on socket events, Application Layer Enforcement (ALE) enforces policies for the secure socket architecture using the Windows Filtering Platform. For more information, see [About Windows Filtering Platform](#) and [Application Layer Enforcement \(ALE\)](#).

## Related topics

[About Windows Filtering Platform](#)

[Advanced Winsock Samples Using Secure Socket Extensions](#)

[Application Layer Enforcement \(ALE\)](#)

[IPsec Configuration](#)

[IPsec Functions](#)

[Secure Winsock Programming](#)

[Security Support Provider Interface \(SSPI\)](#)

[Using Secure Socket Extensions](#)

[Windows Filtering Platform](#)

[Windows Filtering Platform API Functions](#)

# Using Secure Socket Extensions

3/5/2021 • 4 minutes to read • [Edit Online](#)

The following example code demonstrates the usage of the Winsock secure socket extension functions.

## Securing a Socket

```
#define WIN32_LEAN_AND_MEAN

#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>
#include <winsock2.h>
#include <mstcpip.h>
#include <ws2tcpip.h>
#include <rpc.h>
#include <ntdsapi.h>
#include <stdio.h>
#include <tchar.h>

#define RECV_DATA_BUF_SIZE 256

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

// link with fwpclnt.lib for Winsock secure socket extensions
#pragma comment(lib, "fwpclnt.lib")

// link with ntdsapi.lib for DsMakeSpn function
#pragma comment(lib, "ntdsapi.lib")

// The following function assumes that Winsock
// has already been initialized

int
SecureTcpConnect(IN const struct sockaddr *serverAddr,
                 IN ULONG serverAddrLen,
                 IN const wchar_t * serverSPN,
                 IN const SOCKET_SECURITY_SETTINGS * securitySettings,
                 IN ULONG settingsLen)
{**

Routine Description:

    This routine creates a TCP client socket, securely connects to the
    specified server, sends & receives data from the server, and then closes
    the socket

Arguments:

    serverAddr - a pointer to the sockaddr structure for the server.

    serverAddrLen - length of serverAddr in bytes

    serverSPN - a NULL-terminated string representing the SPN
                (service principal name) of the server host computer

    securitySettings - pointer to the socket security settings that should be
                      applied to the connection
```

```

serverAddrLen - length of securitySettings in bytes

Return Value:

Winsock error code indicating the status of the operation, or NO_ERROR if
the operation succeeded.

--*/
{
    int iResult = 0;
    int sockErr = 0;
    SOCKET sock = INVALID_SOCKET;

    WSABUF wsaBuf = { 0 };
    char *dataBuf = "12345678";
    DWORD bytesSent = 0;
    char recvBuf[RECV_DATA_BUF_SIZE] = { 0 };

    DWORD bytesRecv = 0;
    DWORD flags = 0;
    SOCKET_PEER_TARGET_NAME *peerTargetName = NULL;
    DWORD serverSpnStringLen = (DWORD) wcslen(serverSPN);
    DWORD peerTargetNameLen = sizeof(SOCKET_PEER_TARGET_NAME) +
        (serverSpnStringLen * sizeof(wchar_t));

    //-----
    // Create a TCP socket
    sock = WSASocket(serverAddr->sa_family,
                      SOCK_STREAM, IPPROTO_TCP, NULL, 0, 0);
    if (sock == INVALID_SOCKET) {
        iResult = WSAGetLastError();
        wprintf(L"WSASocket returned error %ld\n", iResult);
        goto cleanup;
    }
    //-----
    // Turn on security for the socket.
    sockErr = WSASetSocketSecurity(sock,
                                   securitySettings, settingsLen, NULL, NULL);
    if (sockErr == SOCKET_ERROR) {
        iResult = WSAGetLastError();
        wprintf(L"WSASetSocketSecurity returned error %ld\n", iResult);
        goto cleanup;
    }
    //-----
    // Specify the server SPN
    peerTargetName = (SOCKET_PEER_TARGET_NAME *) HeapAlloc(GetProcessHeap(),
                                           HEAP_ZERO_MEMORY, peerTargetNameLen);
    if (!peerTargetName) {
        iResult = ERROR_NOT_ENOUGH_MEMORY;
        wprintf(L"Out of memory\n");
        goto cleanup;
    }
    // Use the security protocol as specified by the settings
    peerTargetName->SecurityProtocol = securitySettings->SecurityProtocol;
    // Specify the server SPN
    peerTargetName->PeerTargetNameStringLen = serverSpnStringLen;
    RtlCopyMemory((BYTE *) peerTargetName->AllStrings,
                  (BYTE *) serverSPN, serverSpnStringLen * sizeof(wchar_t)
                  );

    sockErr = WSASetSocketPeerTargetName(sock,
                                         peerTargetName,
                                         peerTargetNameLen, NULL, NULL);
    if (sockErr == SOCKET_ERROR) {
        iResult = WSAGetLastError();
        wprintf(L"WSASetSocketPeerTargetName returned error %ld\n", iResult);
        goto cleanup;
    }
    //-----

```

```

// Connect to the server
sockErr = WSACreateSocket(AF_INET, SOCK_STREAM, 0, &sock);
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(443);
serverAddr.sin_addr.s_addr = inet_addr("192.168.1.10");
sockErr = WSAConnect(sock,
                     &serverAddr, sizeof(serverAddr), NULL, NULL, NULL, NULL);
if (sockErr == SOCKET_ERROR) {
    iResult = WSAGetLastError();
    wprintf(L"WSACreateSocket returned error %d\n", iResult);
    goto cleanup;
}
// At this point a secure connection must have been established.
wprintf(L"Secure connection established to the server\n");

//-----
// Send some data securely
wsaBuf.len = (ULONG) strlen(dataBuf);
wsaBuf.buf = dataBuf;
sockErr = WSASend(sock, &wsaBuf, 1, &bytesSent, 0, NULL, NULL);
if (sockErr == SOCKET_ERROR) {
    iResult = WSAGetLastError();
    wprintf(L"WSASend returned error %d\n", iResult);
    goto cleanup;
}
wprintf(L"Sent %d bytes of data to the server\n", bytesSent);

//-----
// Receive server's response securely
wsaBuf.len = RECV_DATA_BUF_SIZE;
wsaBuf.buf = recvBuf;
sockErr = WSARecv(sock, &wsaBuf, 1, &bytesRecv, &flags, NULL, NULL);
if (sockErr == SOCKET_ERROR) {
    iResult = WSAGetLastError();
    wprintf(L"WSARecv returned error %d\n", iResult);
    goto cleanup;
}
wprintf(L"Received %d bytes of data from the server\n", bytesRecv);

cleanup:
if (sock != INVALID_SOCKET) {
    //This will trigger the cleanup of all IPsec filters and policies that
    //were added for this socket. The cleanup will happen only after all
    //outstanding data has been sent out on the wire.
    closesocket(sock);
}
if (peerTargetName) {
    HeapFree(GetProcessHeap(), 0, peerTargetName);
}
return iResult;
}

```

## Querying the Security on a Socket

```

#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <winsock2.h>
#include <mstcpip.h>
#include <ws2tcpip.h>
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

```

```

// link with fwpclnt.lib for Winsock secure socket extensions
#pragma comment(lib, "fwpclnt.lib")

#define MALLOC(x) HeapAlloc(GetProcessHeap(), 0, (x))
#define FREE(x) HeapFree(GetProcessHeap(), 0, (x))

/* Note: could also use malloc() and free() */

// The following function assumes that Winsock
// has already been initialized

int QueryTcpSocketSecurity(IN SOCKET sock)
{
    int iResult = 0;
    SOCKET_SECURITY_QUERY_INFO *queryInfo = NULL;
    ULONG infoLength = 0;

    //First query the number of bytes to allocate
    iResult = WSAQuerySocketSecurity(sock, NULL, NULL, NULL, &infoLength, NULL, NULL);

    if ((iResult == SOCKET_ERROR) && (WSAGetLastError() == WSAEFAULT)) {
        //Allocate the space
        queryInfo = (SOCKET_SECURITY_QUERY_INFO *) MALLOC(infoLength);
        if (!queryInfo) {
            wprintf(L"Unable to allocate memory\n");
            return 1;
        }
    } else {
        wprintf(L"WSAQuerySocketSecurity failed with %u\n", WSAGetLastError());
        return 1;
    }

    //Get the IPsec info
    iResult = WSAQuerySocketSecurity(sock, NULL, NULL, queryInfo, &infoLength, NULL, NULL);

    if (iResult) {
        wprintf(L"WSAQuerySocketSecurity failed with %u\n", WSAGetLastError());
        if (queryInfo)
            FREE(queryInfo);
        return 1;
    }

    // Now queryInfo contains various pieces of information about the
    // security applied to the socket. For instance a server application
    // will be able to access the client access token and impersonate
    // the client if it wants.

    // Note that the SecurityQueryTemplate parameter passed to the
    // WSAQuerySocketSecurity function would need to have the
    // PeerTokenAccessMask member set in the
    // SOCKET_SECURITY_QUERY_TEMPLATE struct to request
    // the return of the PeerApplicationAccessTokenHandle or
    // PeerMachineAccessTokenHandle members in the queryInfo struct
    // returned by WSAQuerySocketSecurity

    // When done, free the memory allocation and exit
    if (queryInfo)
        FREE(queryInfo);
    return iResult;
}

```

## Related topics

[About Windows Filtering Platform](#)

[Advanced Winsock Samples Using Secure Socket Extensions](#)

[Application Layer Enforcement \(ALE\)](#)

[IPsec Configuration](#)

[IPsec Functions](#)

[Security Support Provider Interface \(SSPI\)](#)

[Windows Filtering Platform](#)

[Windows Filtering Platform API Functions](#)

[Winsock Secure Socket Extensions](#)

# Advanced Winsock Samples Using Secure Socket Extensions

3/5/2021 • 2 minutes to read • [Edit Online](#)

## Secure TCP Client and Server Sample

A more advanced Winsock sample that demonstrates the use of secure socket extensions is included with the Microsoft Windows Software Development Kit (SDK). The sample includes a TCP client and server that connect securely using the Winsock and the secure socket extensions.

By default, the Winsock sample source code is installed in the following directory:

*C:\Program Files\Microsoft SDKs\Windows\v6.0\Samples\NetDs\winsock*

A sample is located under the following folder:

***securesocket***

The sample code is split into separate directories as described below:

- **stcpclient** - the folder that contains the secure TCP client code.
- **stcpcommon** - the folder that contains common library code that is shared between the secure TCP client and server.
- **stcpserver** - the folder that contains the secure TCP server code.

It should be noted that the samples are meant to be run on two different computers running Windows Vista or later. Additionally, IPsec credentials must be provisioned on both the computers for the connection to succeed, because the sample uses IPsec for securing its traffic. Please refer to the documentation on [IPsec Configuration](#) for more information on setting up IPsec credentials.

Building the sample will generate two executable files:

***stcpclient.exe*** and ***stcpserver.exe***.

Copy *stcpclient.exe* to computer A and copy *stcpserver.exe* to computer B. On computer B, start the TCP server by executing the following in a Command Prompt:

***stcpserver.exe***

Execute the following command for more usage options for the server:

***stcpserver.exe /?***

Then on computer A, start the TCP client by executing the following in a Command Prompt:

***stcpclient.exe***

At this point the connection should get established securely.

Execute the following command for more usage options for the client:

***stcpclient.exe /?***

## Related topics

[About Windows Filtering Platform](#)

[Application Layer Enforcement \(ALE\)](#)

[IPsec Configuration](#)

[IPsec Functions](#)

[Using Secure Socket Extensions](#)

[Security Support Provider Interface \(SSPI\)](#)

[Windows Filtering Platform](#)

[Windows Filtering Platform API Functions](#)

[Winsock Secure Socket Extensions](#)

# Porting Socket Applications to Winsock

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section describes Winsock porting considerations.

There are a limited number of instances where Windows Sockets has diverted from strict adherence to the Berkeley conventions, usually due to implementation difficulties in the Microsoft Windows environment.

When a deviation from Berkeley conventions occurs in Windows Sockets, the deviation is specifically and clearly noted. For example, if a function is specific to Windows Sockets, that deviation is specified with a phrase in the function description similar to the following:

The [function-name] function is a Microsoft-specific extension to the Windows Sockets 2 API.

This section provides information about porting Berkeley (BSD) UNIX socket applications to Winsock:

- [Socket Data Type](#)
- [Select, FD\\_SET, and FD\\_XXX Macros](#)
- [Error Codes - errno, h\\_errno and WSAGetLastError](#)
- [Pointers](#)
- [Renamed Functions](#)
- [Maximum Number of Sockets Supported](#)
- [Include Files](#)
- [Return Values on Function Failure](#)
- [Raw Sockets](#)
- [Byte Ordering](#)
- [Extended Byte-Order Conversion Routines](#)

## Related topics

[Handling Winsock Errors](#)

[Porting Socket Applications to Winsock](#)

[Windows Sockets Error Codes](#)

[Winsock Programming Considerations](#)

# Socket Data Type

3/5/2021 • 2 minutes to read • [Edit Online](#)

In Winsock applications, a socket descriptor is not a file descriptor and must be used with the Winsock functions.

In UNIX, a socket descriptor is represented by a standard file descriptor. As a result, a socket descriptor on UNIX may be passed to any of the standard file I/O functions (read and write, for example).

Furthermore, all handles in UNIX, 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** functions 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 UNIX). Instead, an application should use the manifest constant INVALID\_SOCKET as defined in the *Winsock2.h* header file. For example:

Typical BSD UNIX Style

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

Preferred Style

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

## Related topics

[Porting Socket Applications to Winsock](#)

[Winsock Programming Considerations](#)

# Select, FD\_SET, and FD\_XXX Macros

3/5/2021 • 2 minutes to read • [Edit Online](#)

Since sockets are not 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](#) structure, but instead of being stored as a bitmask, the set is implemented as an array of sockets. To avoid potential problems, applications must adhere to the use of the FD\_XXX macros to set, initialize, clear, and check the [FD\\_SET](#) structures.

## Related topics

[Porting Socket Applications to Winsock](#)

[Winsock Programming Considerations](#)

# Error Codes - `errno`, `h_errno` and `WSAGetLastError`

3/5/2021 • 3 minutes to read • [Edit Online](#)

In Winsock applications, error codes are retrieved using the `WSAGetLastError` function, the Windows Sockets substitute for the Windows `GetLastError` function. The error codes returned by Windows Sockets are similar to UNIX socket error code constants, but the constants are all prefixed with WSA. So in Winsock applications the `WSAEWOULDBLOCK` error code would be returned, while in UNIX applications the `EWOULDBLOCK` error code would be returned.

Error codes set by Windows Sockets are not made available through the `errno` variable. Additionally, for the `getXbyY` class of functions, error codes are not made available through the `h_errno` variable. The `WSAGetLastError` function is intended to provide a reliable way for a thread in a multithreaded process to obtain per-thread error information.

For compatibility with Berkeley UNIX (BSD), early versions of Windows (Windows 95 with the Windows Socket 2 Update and Windows 98, for example) redefined regular Berkeley error constants typically found in `errno.h` on BSD as the equivalent Windows Sockets WSA errors. So for example, `ECONNREFUSED` was defined as `WSAECONNREFUSED` in the `Winsock.h` header file. In subsequent versions of Windows (Windows NT 3.1 and later) these defines were commented out to avoid conflicts with `errno.h` used with Microsoft C/C++ and Visual Studio.

The `Winsock2.h` header file included with the Microsoft Windows Software Development Kit (SDK), Platform Software Development Kit (SDK), and Visual Studio still contains a commented out block of defines within an `#ifdef 0` and `#endif` block that define the BSD socket error codes to be the same as the WSA error constants. These can be used to provide some compatibility with UNIX, BSD, and Linux socket programming. For compatibility with BSD, an application may choose to change the `Winsock2.h` and uncomment this block. However, application developers are strongly discouraged from uncommenting this block because of inevitable conflicts with `errno.h` in most applications. Also, the BSD socket errors are defined to very different values than are used in UNIX, BSD, and Linux programs. Application developers are very strongly encouraged to use the WSA error constants in socket applications.

These defines remain commented out in the `Winsock2.h` header within an `#ifdef 0` and `#endif` block. If an application developer insists on using the BSD error codes for compatibility, then an application may choose to include a line of the form:

```
#include <windows.h>

#define errno WSAGetLastError()
```

This allows networking code which was written to use the global `errno` to work correctly in a single-threaded environment. There are some very serious 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 Berkeley Sockets 4.3 are provided for compatibility purposes, applications are strongly encouraged to use the WSA error code definitions. This is because error codes returned by certain Windows Sockets functions fall into the standard range of error codes as defined by Microsoft C<sup>©</sup>. Thus, a better version of the preceding source code fragment is:

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

The original Winsock 1.1 specification defined in 1995 recommended a set of error codes, and listed the possible errors that can be returned as a result of each function. Windows Sockets 2 added functions and features with other Windows Sockets error codes returned in addition to those listed in the original Winsock specification. Additional functions have been added over time to enhance Winsock for use by developers. For example, new name service functions ([getaddrinfo](#) and [getnameinfo](#), for example) were added that support both IPv6 and IPv4 on Windows XP and later. Some of the older IPv4-only name service functions (the [getXbyY](#) class of functions, for example) have been deprecated.

A complete list of possible error codes returned by Windows Sockets functions is given in the section on [Windows Sockets Error Codes](#).

## Related topics

[Handling Winsock Errors](#)

[Porting Socket Applications to Winsock](#)

[Windows Sockets Error Codes](#)

[Winsock Programming Considerations](#)

# Pointers (Windows Sockets 2)

3/5/2021 • 2 minutes to read • [Edit Online](#)

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.

## Related topics

[hostent](#)

[Porting Socket Applications to Winsock](#)

[Winsock Programming Considerations](#)

# Renamed Functions

3/5/2021 • 2 minutes to read • [Edit Online](#)

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

## Close and 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 Windows Sockets prevents an implementation from using regular file handles to identify sockets, nothing requires it either. On Windows, sockets must be closed by using the **closesocket** routine. ON Windows, using the **close** function to close a socket is incorrect and the effects of doing so are undefined by this specification.

## Ioctl and Ioctlsocket/WSAIoctl

Various C language run-time systems use the IOCTLs 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.

## Related topics

[closesocket](#)

[ioctlsocket](#)

[Porting Socket Applications to Winsock](#)

[Winsock Programming Considerations](#)

[WSAIoctl](#)

# Maximum Number of Sockets Supported

3/5/2021 • 2 minutes to read • [Edit Online](#)

The maximum number of sockets supported by a particular Windows Sockets service provider is implementation specific. The Microsoft Winsock provider limits the maximum number of sockets supported only by available memory on the local computer. However, third-party Winsock providers may have limitations on the numbers of sockets supported. An application should make no assumptions about the availability of a certain number of sockets. For more information on this topic see [WSAStartup](#).

## FD\_SET and select

A number of FD\_XXX macros are defined in the *Winsock2.h* header file for use in porting applications to Windows from the UNIX environment. These macros are used with the **select** and **WSAPoll** functions for porting applications to Windows. The maximum number of sockets that a Windows Sockets application can use is not affected by the manifest constant FD\_SETSIZE. This value defined in the *Winsock2.h* header file is used in constructing the **FD\_SET** structures used with **select** function. The default value in *Winsock2.h* is 64. If an application is designed to be capable of working with more than 64 sockets using the **select** and **WSAPoll** functions, the implementor should define the manifest FD\_SETSIZE in every source file before including the *Winsock2.h* header file. 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. This value only affects the FD\_XXX macros used by the **select** and **WSAPoll** functions.

## Related topics

[FD\\_SET](#)

[Porting Socket Applications to Winsock](#)

[select](#)

[Winsock Programming Considerations](#)

[WSAStartup](#)

[WSAPoll](#)

# Include Files

3/5/2021 • 2 minutes to read • [Edit Online](#)

The original include file for use with Windows Sockets 1.1 was the *Winsock.h* header file. To ease porting existing source code based on Berkeley UNIX sockets to Windows sockets, Windows Sockets development kits for Winsock 1.1 were encouraged to be supplied with several include files with the same names as standard UNIX include files (the *sys/socket.h* and *arpa/inet.h* header files, for example). However, these similarly-name Winsock header files merely contained a directive to include the *Winsock2.h* header file.

When Windows Sockets 2 was released, the primary include file for use with Windows Sockets was renamed to *Winsock2.h*. The older original *Winsock.h* header file for Winsock 1.1 was also retained for compatibility with older applications. The development of Winsock 1.1 compatible applications has been deprecated since Windows 2000 was released. All applications should now use the include *Winsock2.h* directive in Winsock application source files.

The *Winsock2.h* header file contains most of the Winsock functions, structures, and definitions. The *Ws2tcpip.h* header file contains definitions introduced in the WinSock 2 Protocol-Specific Annex document for TCP/IP that includes newer functions and structures used to retrieve IP addresses. These include the [getaddrinfo](#) and [getnameinfo](#) family of functions that provide name resolution for both IPv4 or IPv6 addresses. The *Ws2tcpip.h* header file is only needed if these IP-agnostic naming functions are required by the application.

The *Mswsock.h* header file contains definitions for Microsoft-specific extensions to the Windows Sockets 2 ([TransmitFile](#), [AcceptEx](#), and [ConnectEx](#), for example). The *Mswsock.h* header file is not normally needed unless these Microsoft-specific extensions are used by the application.

The *Winsock2.h* header file internally includes core elements from the *Windows.h* header file, so there is not usually an #include line for the *Windows.h* header file in Winsock applications. If an #include line is needed for the *Windows.h* header file, this should be preceded with the #define WIN32\_LEAN\_AND\_MEAN macro. For historical reasons, the *Windows.h* header defaults to including the *Winsock.h* header file for Windows Sockets 1.1. The declarations in the *Winsock.h* header file will conflict with the declarations in the *Winsock2.h* header file required by Windows Sockets 2. The WIN32\_LEAN\_AND\_MEAN macro prevents the *Winsock.h* from being included by the *Windows.h* header. An example illustrating this is shown below.

```
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

#pragma comment(lib, "Ws2_32.lib")

int main() {
    return 0;
}
```

## Related topics

[Creating a Basic Winsock Application](#)

[Getting Started With Winsock](#)

[Porting Socket Applications to Winsock](#)

[Winsock Programming Considerations](#)



# Return Values on Function Failure

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 (will not work on Windows)

```
r = recv(ClientSocket, recvbuf, recvbuflen, 0);
if (r == -1      /* or r < 0 */
    && errno == EWOULDBLOCK) {
    printf("recv failed with error: EWOULDBLOCK\n");
}
```

Windows Style

```
iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);
if (iResult == SOCKET_ERROR ) {
    iError = WSAGetLastError();
    if (iError == WSAEWOULDBLOCK)
        printf("recv failed with error: WSAEWOULDBLOCK\n");
    else
        printf("recv failed with error: %ld\n", iError);

    closesocket(ClientSocket);
    WSACleanup();
    return 1;
}
```

## Related topics

[Error Codes - errno, h\\_errno and WSAGetLastError](#)

[Handling Winsock Errors](#)

[Porting Socket Applications to Winsock](#)

[Winsock Programming Considerations](#)

[Windows Sockets Error Codes](#)

# Raw Sockets

3/5/2021 • 2 minutes to read • [Edit Online](#)

A raw socket is a type of socket that allows access to the underlying transport provider. The use of raw sockets when porting applications to Winsock is not recommended for several reasons.

The Windows Sockets specification does not mandate that a Winsock 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.

On Windows 7, Windows Server 2008 R2, Windows Vista, and Windows XP with Service Pack 2 (SP2), the ability to send traffic over raw sockets has been restricted in several ways. For more information, see [TCP/IP Raw Sockets](#).

## Related topics

[Porting Socket Applications to Winsock](#)

[socket](#)

[TCP/IP Raw Sockets](#)

[Winsock Programming Considerations](#)

# Byte Ordering

3/5/2021 • 2 minutes to read • [Edit Online](#)

Care must always be taken to account for any differences between the byte ordering used for storing integers by the host 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 (big-endian) for the protocol being utilized. In the case of IP, this includes the IP address and port parameters of a `sockaddr` structure (but not the `sin_family` parameter).

A number of the UNIX systems operate on CPUs that represent integers in network byte order (big-endian). So the need to convert integers from host byte order to network byte order can be ignored without causing problems even if this is not recommended for portability.

In contrast, the byte ordering used to represent integers by most Intel® CPUs is little-endian. So it becomes mandatory that integers be converted from host byte-order to network byte order before being used in Winsock Sockets functions and structures.

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 that 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 `hton` or `WSAHton` 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 the `ntoh` or `WSANtoh` function) before it can be displayed.

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

## Related topics

[getpeername](#)

[getservbyname](#)

[htonl](#)

[htons](#)

[ntohl](#)

[ntohs](#)

[Porting Socket Applications to Winsock](#)

[sockaddr](#)

[Winsock Programming Considerations](#)

[WSAHtonl](#)

[WSAHtons](#)

**WSANtohl**

**WSANtohs**

# Extended Byte-Order Conversion Routines

3/5/2021 • 2 minutes to read • [Edit Online](#)

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 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).

## Related topics

[htonl](#)

[htons](#)

[ntohl](#)

[ntohs](#)

[Porting Socket Applications to Winsock](#)

[Winsock Programming Considerations](#)

[WSAHtonl](#)

[WSAHtons](#)

[WSANtohl](#)

[WSANtohs](#)

[WSAPROTOCOL\\_INFO](#)

# IPv6 Guide for Windows Sockets Applications

3/5/2021 • 2 minutes to read • [Edit Online](#)

This guide provides the information you need to enable your Microsoft Windows application to use the next generation of Internet Protocol, version 6 (IPv6). Adding IPv6 capability to your application is not necessarily a porting process. To port an application suggests modifying code to work on a different platform, which implies leaving the previous platform behind. This guide is specifically structured to help add IPv6 capability to an application while retaining IPv4 functionality.

This guide discusses the issues associated with adding IPv6 functionality, then targets the areas of development most affected by the transition. Each area receives a thorough explanation of the pitfalls to watch out for, the strategies suggested to avoid them, and tips on how to make the best use of new [Windows Sockets 2](#) programmatic elements (functions and structures). For additional information on IPv6, see [IPv6 Support](#).

This guide also provides code examples to give you hands-on experience and visual representations of the issues you could encounter when modifying your applications. The examples come from complete, working examples of a simple Windows Sockets application that has been modified to support both IPv4 and IPv6. Source code for these working examples is included in its entirety in two appendixes at the end of this document: [Appendix A: IPv4-only Source Code](#) includes the source code for an application before it is modified to support IPv6; [Appendix B: IP-version Agnostic Source Code](#) provides the source code after the application has been IPv6-enabled.

Microsoft provides a utility called Checkv4.exe that helps you find potentially porting-sensitive code in your application code, and also makes recommendations for fixes. The Checkv4.exe utility is demonstrated in this document, using the sample application included in the appendixes, along with screen shots displaying the output that the Checkv4.exe utility produces. For more information, see [Using the Checkv4.exe Utility](#).

The programming areas addressed by this guide are:

- [Changing Data Structures for IPv6 Winsock Applications](#)
- [Function Calls for IPv6 Winsock Applications](#)
- [Use of Hardcoded IPv4 Addresses](#)
- [User Interface Issues for IPv6 Winsock Applications](#)
- [Underlying Protocols for IPv6 Winsock Applications](#)
- [Dual-Stack Sockets for IPv6 Winsock Applications](#)

Because there is no uniform sequence of events, the sections that address IPv6-enabling issues are not arranged in a sequentially significant manner, so you can reference any section at any time. It is strongly recommended that you review each section while adding IPv6 capability to your application. It is also advisable to read about the Checkv4.exe utility, as it includes tips on the order in which to address IPv6-enabling issues.

To look at the Checkv4.exe utility, and to review the order in which you should approach the porting process in your applications, see [Using the Checkv4.exe Utility](#). This section includes information on a compile-time flag that strictly checks for programming elements incompatible with IPv6.

To go straight to the sample application, see [Appendix A: IPv4-only Source Code](#) and [Appendix B: IP-version Agnostic Source Code](#).

## Related topics

[Internet Protocol Version 6 \(IPv6\)](#)

[IPv6 Support](#)

[IPv6 Technology Preview for Windows 2000](#)

[Using the Checkv4.exe Utility](#)

[Appendix A: IPv4-only Source Code](#)

[Appendix B: IP-version Agnostic Source Code](#)

# Changing Data Structures for IPv6 Winsock Applications

3/5/2021 • 2 minutes to read • [Edit Online](#)

When adding support for IPv6, you must ensure that your application defines properly sized data structures. The size of an IPv6 address is much larger than an IPv4 address. Structures that are hard-coded to handle the size of an IPv4 address when storing an IP address will cause problems in your application, and must be modified.

## Best Practice

The best approach to ensuring that your structures are properly sized is to use the **SOCKADDR\_STORAGE** structure. The **SOCKADDR\_STORAGE** structure is agnostic to IP address version. When the **SOCKADDR\_STORAGE** structure is used to store IP addresses, IPv4 and IPv6 addresses can be properly handled with one code base.

The following example, which is an excerpt taken from the Server.c file found in Appendix B, identifies an appropriate use of the **SOCKADDR\_STORAGE** structure. Notice that the structure, when used properly as this example shows, gracefully handles either an IPv4 or IPv6 address.

```

#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>

#pragma comment(lib, "Ws2_32.lib")

#define BUFFER_SIZE 512
#define DEFAULT_PORT "27015"

int main(int argc, char **argv)
{
    char Buffer[BUFFER_SIZE] = {0};
    char *Hostname;
    int Family = AF_UNSPEC;
    int SocketType = SOCK_STREAM;
    char *Port = DEFAULT_PORT;
    char *Address = NULL;
    int i = 0;
    DWORD dwRetval = 0;
    int iResult = 0;
    int FromLen = 0;
    int AmountRead = 0;

    SOCKADDR_STORAGE From;

    WSADATA wsaData;

    ADDRINFO *AddrInfo = NULL;
    ADDRINFO *AI = NULL;

    // Parse arguments
    if (argc >= 1) {
        Hostname = argv[1];
    }

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        printf("WSAStartup failed: %d\n", iResult);
        return 1;
    }

    From.ss_family = (ADDRESS_FAMILY) Family;

    //...

    return 0;
}

```

#### **NOTE**

The **SOCKADDR\_STORAGE** structure is new for Windows XP.

#### Code To Avoid

Typically, many applications used the **sockaddr** structure to store protocol-independent addresses, or the **sockaddr\_in** structure for IP addresses. Neither the **sockaddr** structure nor the **sockaddr\_in** structure is large enough to hold IPv6 addresses, and therefore both are insufficient if your application is to be IPv6 compatible.

#### Coding Task

To modify your existing code base from IPv4 to IPv4- and IPv6-interoperability

1. Acquire the Checkv4.exe utility. The utility is included with the Microsoft Windows Software Development Kit (SDK) which is made available through your MSDN subscription, or from the web as a download.
2. Run the Checkv4.exe utility against your code. Learn about how to run the Checkv4.exe utility against your files in the section on [Using the Checkv4.exe Utility](#).
3. The utility alerts you to usage of `sockaddr` or `sockaddr_in` structures, and provides recommendations on how to replace either with the IPv6 compatible structure `SOCKADDR_STORAGE`.
4. Replace any such instances, and associated code as appropriate, to use the `SOCKADDR_STORAGE` structure.

Alternatively, you can search your code base for instances of the `sockaddr` and `sockaddr_in` structures, and change all such usage (and other associated code, as appropriate) to the `SOCKADDR_STORAGE` structure.

**NOTE**

The `addrinfo` and `SOCKADDR_STORAGE` structures include protocol and address family members (`ai_family` and `ss_family`), respectively. RFC 2553 specifies the `ai_family` member of `addrinfo` as an int, while `ss_family` is specified as a short; as such, a direct copy between those members results in a compiler error.

## Related topics

[IPv6 Guide for Windows Sockets Applications](#)

[Dual-Stack Sockets for IPv6 Winsock Applications](#)

[Function Calls for IPv6 Winsock Applications](#)

[Use of Hardcoded IPv4 Addresses](#)

[User Interface Issues for IPv6 Winsock Applications](#)

[Underlying Protocols for IPv6 Winsock Applications](#)

# Function Calls for IPv6 Winsock Applications

3/5/2021 • 13 minutes to read • [Edit Online](#)

New functions have been introduced to the Windows Sockets interface specifically designed to make Windows Sockets programming easier. One of the benefits of these new Windows Sockets functions is integrated support for both IPv6 and IPv4.

These new Windows Sockets functions include the following:

- [WSAConnectByName](#)
- [WSAConnectByList](#)
- [getaddrinfo](#) family of functions ([getaddrinfo](#), [GetAddrInfoEx](#), [GetAddrInfoW](#), [freeaddrinfo](#), [FreeAddrInfoEx](#), [FreeAddrInfoW](#), and [SetAddrInfoEx](#))
- [getnameinfo](#) family of functions ([getnameinfo](#) and [GetNameInfoW](#))

In addition, new IP Helper functions with support for both IPv6 and IPv4 have been added to simplify Windows Sockets programming. These new IP Helper functions include the following:

- [GetAdaptersAddresses](#)

When adding IPv6 support to an application the following guidelines should be used:

- Use [WSAConnectByName](#) to establish a connection to an endpoint given a host name and port. The [WSAConnectByName](#) function is available on Windows Vista and later.
- Use [WSAConnectByList](#) to establish a connection to one out of a collection of possible endpoints represented by a set of destination addresses (host names and ports). The [WSAConnectByList](#) function is available on Windows Vista and later.
- Replace [gethostbyname](#) function calls with calls to one of the new [getaddrinfo](#) Windows Sockets functions. The [getaddrinfo](#) function with support for the IPv6 protocol is available on Windows XP and later. The IPv6 protocol is also supported on Windows 2000 when the IPv6 Technology Preview for Windows 2000 is installed.
- Replace [gethostbyaddr](#) function calls with calls to one of the new [getnameinfo](#) Windows Sockets functions. The [getnameinfo](#) function with support for the IPv6 protocol is available on Windows XP and later. The IPv6 protocol is also supported on Windows 2000 when the IPv6 Technology Preview for Windows 2000 is installed.

## WSAConnectByName

The [WSAConnectByName](#) function simplifies connecting to an endpoint using a stream-based socket given the destination's hostname or IP address (IPv4 or IPv6). This function reduces the source code required to create an IP application that is agnostic to the version of the IP protocol used. [WSAConnectByName](#) replaces the following steps in a typical TCP application to a single function call:

- Resolve a hostname to a set of IP addresses.
- For each IP address:
  - Create a socket of the appropriate address family.
  - Attempts to connect to the remote IP address. If the connection was successful, it returns; otherwise the next remote IP address for the host is tried.

The [WSAConnectByName](#) function goes beyond just resolving the name and then attempting to connect. The function uses all of the remote addresses returned by name resolution and all of the local machine's source IP

addresses. It first attempts to connect using address pairs with the highest chance of success. Therefore, **WSAConnectByName** not only ensures that a connection will be established if possible, but it also minimizes the time to establish the connection.

To enable both IPv6 and IPv4 communications, use the following method:

- The **setsockopt** function must be called on a socket created for the AF\_INET6 address family to disable the IPV6\_V6ONLY socket option before calling **WSAConnectByName**. This is accomplished by calling the **setsockopt** function on the socket with the *level* parameter set to IPPROTO\_IPV6 (see [IPPROTO\\_IPV6 Socket Options](#)), the *optname* parameter set to IPV6\_V6ONLY, and the *optvalue* parameter value set to zero.

If an application needs to bind to a specific local address or port, then **WSAConnectByName** cannot be used since the socket parameter to **WSAConnectByName** must be an unbound socket.

The code example below shows only a few lines of code are needed to use this function to implement an application that is agnostic to the IP version.

Establish a connection using **WSAConnectByName**

```

#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

SOCKET OpenAndConnect(LPWSTR NodeName, LPWSTR PortName)
{
    SOCKET ConnSocket;
    DWORD ipv6only = 0;
    int iResult;
    BOOL bSuccess;
    SOCKADDR_STORAGE LocalAddr = {0};
    SOCKADDR_STORAGE RemoteAddr = {0};
    DWORD dwLocalAddr = sizeof(LocalAddr);
    DWORD dwRemoteAddr = sizeof(RemoteAddr);

    ConnSocket = socket(AF_INET6, SOCK_STREAM, 0);
    if (ConnSocket == INVALID_SOCKET){
        return INVALID_SOCKET;
    }

    iResult = setsockopt(ConnSocket, IPPROTO_IPV6,
        IPV6_V6ONLY, (char*)&ipv6only, sizeof(ipv6only) );
    if (iResult == SOCKET_ERROR){
        closesocket(ConnSocket);
        return INVALID_SOCKET;
    }

    bSuccess = WSAConnectByName(ConnSocket, NodeName,
        PortName, &dwLocalAddr,
        (SOCKADDR*)&LocalAddr,
        &dwRemoteAddr,
        (SOCKADDR*)&RemoteAddr,
        NULL,
        NULL);
    if (bSuccess){
        return ConnSocket;
    } else {
        return INVALID_SOCKET;
    }
}

```

## WSAConnectByList

The [WSAConnectByList](#) function establishes a connection to a host given a set of possible hosts (represented by a set of destination IP addresses and ports). The function takes all IP addresses and ports for the endpoint and all of the local machine's IP addresses, and attempts a connection using all possible address combinations.

[WSAConnectByList](#) is related to the [WSAConnectByName](#) function. Instead of taking a single hostname, [WSAConnectByList](#) accepts a list of hosts (destination addresses and port pairs) and connects to one of the addresses and ports in the provided list. This function is designed to support scenarios in which an application needs to connect to any available host out of a list of potential hosts.

Similar to [WSAConnectByName](#), the [WSAConnectByList](#) function significantly reduces the source code required to create, bind and connect a socket. This function makes it much easier to implement an application

that is agnostic to the IP version. The list of addresses for a host accepted by this function may be IPv6 or IPv4 addresses.

To enable both IPv6 and IPv4 addresses to be passed in the single address list accepted by the function, the following steps must be performed prior to calling the function:

- The [setsockopt](#) function must be called on a socket created for the AF\_INET6 address family to disable the IPV6\_V6ONLY socket option before calling [WSAConnectByList](#). This is accomplished by calling the [setsockopt](#) function on the socket with the *level* parameter set to [IPPROTO\\_IPV6](#) (see [IPPROTO\\_IPV6 Socket Options](#)), the *optname* parameter set to [IPV6\\_V6ONLY](#), and the *optvalue* parameter value set to zero.
- Any IPv4 addresses must be represented in the IPv4-mapped IPv6 address format which enables an IPv6 only application to communicate with an IPv4 node. The IPv4-mapped IPv6 address format allows the IPv4 address of an IPv4 node to be represented as an IPv6 address. The IPv4 address is encoded into the low-order 32 bits of the IPv6 address, and the high-order 96 bits hold the fixed prefix 0:0:0:0:FFFF. The IPv4-mapped IPv6 address format is specified in RFC 4291. For more information, see [www.ietf.org/rfc/rfc4291.txt](http://www.ietf.org/rfc/rfc4291.txt). The IN6ADDR\_SETV4MAPPED macro in *Mstcpip.h* can be used to convert an IPv4 address to the required IPv4-mapped IPv6 address format.

Establish a Connection Using [WSAConnectByList](#)

```

#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

SOCKET OpenAndConnect(SOCKET_ADDRESS_LIST *AddressList)
{
    SOCKET ConnSocket;
    DWORD ipv6only = 0;
    int iResult;
    BOOL bSuccess;
    SOCKADDR_STORAGE LocalAddr = {0};
    SOCKADDR_STORAGE RemoteAddr = {0};
    DWORD dwLocalAddr = sizeof(LocalAddr);
    DWORD dwRemoteAddr = sizeof(RemoteAddr);

    ConnSocket = socket(AF_INET6, SOCK_STREAM, 0);
    if (ConnSocket == INVALID_SOCKET){
        return INVALID_SOCKET;
    }

    iResult = setsockopt(ConnSocket, IPPROTO_IPV6,
        IPV6_V6ONLY, (char*)&ipv6only, sizeof(ipv6only) );
    if (iResult == SOCKET_ERROR){
        closesocket(ConnSocket);
        return INVALID_SOCKET;
    }

    // AddressList may contain IPv6 and/or IPv4Mapped addresses
    bSuccess = WSACConnectByList(ConnSocket,
        AddressList,
        &dwLocalAddr,
        (SOCKADDR*)&LocalAddr,
        &dwRemoteAddr,
        (SOCKADDR*)&RemoteAddr,
        NULL,
        NULL);
    if (bSuccess){
        return ConnSocket;
    } else {
        return INVALID_SOCKET;
    }
}

```

## getaddrinfo

The **getaddrinfo** function also performs the processing work of many functions. Previously, calls to a number of Windows Sockets functions were required to create, open, and then bind an address to a socket. With the **getaddrinfo** function, the lines of source code necessary to perform such work can be significantly reduced. The following two examples illustrate the source code necessary to perform these tasks with and without the **getaddrinfo** function.

Perform an Open, Connect, and Bind Using **getaddrinfo**

```
#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

SOCKET OpenAndConnect(char *ServerName, char *PortName, int SocketType)
{
    SOCKET ConnSocket;
    ADDRINFO *AI;

    if (getaddrinfo(ServerName, PortName, NULL, &AI) != 0) {
        return INVALID_SOCKET;
    }

    ConnSocket = socket(AI->ai_family, SocketType, 0);
    if (ConnSocket == INVALID_SOCKET) {
        freeaddrinfo(AI);
        return INVALID_SOCKET;
    }

    if (connect(ConnSocket, AI->ai_addr, (int) AI->ai_addrlen) == SOCKET_ERROR) {
        closesocket(ConnSocket);
        freeaddrinfo(AI);
        return INVALID_SOCKET;
    }

    return ConnSocket;
}
```

Perform an Open, Connect, and Bind Without Using `getaddrinfo`

```

#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

SOCKET OpenAndConnect(char *ServerName, unsigned short Port, int SocketType)
{
    SOCKET ConnSocket;
    LPHOSTENT hp;
    SOCKADDR_IN ServerAddr;

    ConnSocket = socket(AF_INET, SocketType, 0); /* Open a socket */
    if (ConnSocket < 0 ) {
        return INVALID_SOCKET;
    }

    memset(&ServerAddr, 0, sizeof(ServerAddr));
    ServerAddr.sin_family = AF_INET;
    ServerAddr.sin_port = htons(Port);

    if (isalpha(ServerName[0])) { /* server address is a name */
        hp = gethostbyname(ServerName);
        if (hp == NULL) {
            return INVALID_SOCKET;
        }
        ServerAddr.sin_addr.s_addr = (ULONG) hp->h_addr;
    } else { /* Convert nnn.nnn address to a usable one */
        ServerAddr.sin_addr.s_addr = inet_addr(ServerName);
    }

    if (connect(ConnSocket, (LPSOCKADDR)&ServerAddr,
               sizeof(ServerAddr)) == SOCKET_ERROR)
    {
        closesocket(ConnSocket);
        return INVALID_SOCKET;
    }

    return ConnSocket;
}

```

Notice that both of these source code examples perform the same tasks, but the first example, using the [getaddrinfo](#) function, requires fewer lines of source code, and can handle either IPv6 or IPv4 addresses. The number of lines of source code eliminated by using the [getaddrinfo](#) function varies.

#### **NOTE**

In production source code, your application would iterate through the set of addresses returned by the [gethostbyname](#) or [getaddrinfo](#) function. These examples omit that step in favor of simplicity.

Another issue you must address when modifying an existing IPv4 application to support IPv6 is associated with the order in which functions are called. Both [getaddrinfo](#) and [gethostbyname](#) require that a sequence of function calls are made in a specific order.

On platforms where both IPv4 and IPv6 are used, the address family of the remote host name is not known ahead of time. So address resolution using the [getaddrinfo](#) function must be executed first to determine the IP address and address family of the remote host. Then the [socket](#) function can be called to open a socket of the address family returned by [getaddrinfo](#). This is an important change in how Windows Sockets applications are written, since many IPv4 applications tend to use a different order of function calls.

Most IPv4 applications first create a socket for the AF\_INET address family, then do name resolution, and then use the socket to connect to the resolved IP address. When making such applications IPv6-capable, the [socket](#) function call must be delayed until after name resolution when the address family has been determined. Note that if name resolution returns both IPv4 and IPv6 addresses, then separate IPv4 and IPv6 sockets must be used to connect to these destination addresses. All of these complexities can be avoided by using the [WSAConnectByName](#) function on Windows Vista and later, so application developers are encouraged to use this new function.

The following code example shows the proper sequence for performing name resolution first (performed in the fourth line in the following source code example), then opening a socket (performed in the 19<sup>th</sup> line in the following code example). This example is an excerpt from the Client.c file found in the [IPv6-Enabled Client Code](#) in Appendix B. The PrintError function called in the following code example is listed in the Client.c sample.

```
#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

SOCKET OpenAndConnect(char *Server, char *PortName, int Family, int SocketType)
{

    int iResult = 0;
    SOCKET ConnSocket = INVALID_SOCKET;

    ADDRINFO *AddrInfo = NULL;
    ADDRINFO *AI = NULL;
    ADDRINFO Hints;

    char *AddrName = NULL;

    memset(&Hints, 0, sizeof (Hints));
    Hints.ai_family = Family;
    Hints.ai_socktype = SocketType;

    iResult = getaddrinfo(Server, PortName, &Hints, &AddrInfo);
    if (iResult != 0) {
        printf("Cannot resolve address [%s] and port [%s], error %d: %s\n",
               Server, PortName, WSAGetLastError(), gai_strerror(iResult));
        return INVALID_SOCKET;
    }
    //
    // Try each address getaddrinfo returned, until we find one to which
    // we can successfully connect.
    //
    for (AI = AddrInfo; AI != NULL; AI = AI->ai_next) {

        // Open a socket with the correct address family for this address.
        ConnSocket = socket(AI->ai_family, AI->ai_socktype, AI->ai_protocol);
        if (ConnSocket == INVALID_SOCKET) {
            printf("Error opening socket - error %d\n" WSAGetLastError());
        }
    }
}
```

```

        printf("Error opening socket, error %u\n", WSAGetLastError());
        continue;
    }
}

// Notice that nothing in this code is specific to whether we
// are using UDP or TCP.
//
// When connect() is called on a datagram socket, it does not
// actually establish the connection as a stream (TCP) socket
// would. Instead, TCP/IP establishes the remote half of the
// (LocalIPAddress, LocalPort, RemoteIP, RemotePort) mapping.
// This enables us to use send() and recv() on datagram sockets,
// instead of recvfrom() and sendto().
//

printf("Attempting to connect to: %s\n", Server ? Server : "localhost");
if (connect(ConnSocket, AI->ai_addr, (int) AI->ai_addrlen) != SOCKET_ERROR)
    break;

if (getnameinfo(AI->ai_addr, (socklen_t) AI->ai_addrlen, AddrName,
                sizeof (AddrName), NULL, 0, NI_NUMERICHOST) != 0) {
    strcpy_s(AddrName, sizeof (AddrName), "<unknown>");
    printf("connect() to %s failed with error %d\n", AddrName, WSAGetLastError());
    closesocket(ConnSocket);
    ConnSocket = INVALID_SOCKET;
}
}

return ConnSocket;
}

```

## IP Helper Functions

Finally, applications making use of the IP Helper function [GetAdaptersInfo](#), and its associated structure [IP\\_ADAPTER\\_INFO](#), must recognize that both this function and structure are limited to IPv4 addresses. IPv6-enabled replacements for this function and structure are the [GetAdaptersAddresses](#) function and the [IP\\_ADAPTER\\_ADDRESSES](#) structure. IPv6-enabled applications making use of the IP Helper API should use the [GetAdaptersAddresses](#) function and the corresponding IPv6-enabled [IP\\_ADAPTER\\_ADDRESSES](#) structure, both defined in the Microsoft Windows Software Development Kit (SDK).

## Recommendations

The best approach to ensure your application is using IPv6-compatible function calls is to use the [getaddrinfo](#) function for obtaining host-to-address translation. Beginning with Windows XP, the [getaddrinfo](#) function makes the [gethostbyname](#) function unnecessary, and your application should therefore use the [getaddrinfo](#) function instead for future programming projects. While Microsoft will continue to support [gethostbyname](#), this function will not be extended to support IPv6. For transparent support for obtaining IPv6 and IPv4 host information, you must use [getaddrinfo](#).

The following example shows how to best use the [getaddrinfo](#) function. Notice that the function, when used properly as this example demonstrates, handles both IPv6 and IPv4 host-to-address translation properly, but it also obtains other useful information about the host, such as the type of sockets supported. This sample is an excerpt from the Client.c sample found in Appendix B.

```

#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

int ResolveName(char *Server, char *PortName, int Family, int SocketType)
{
    int iResult = 0;

    ADDRINFO *AddrInfo = NULL;
    ADDRINFO *AI = NULL;
    ADDRINFO Hints;

    //
    // By not setting the AI_PASSIVE flag in the hints to getaddrinfo, we're
    // indicating that we intend to use the resulting address(es) to connect
    // to a service. This means that when the Server parameter is NULL,
    // getaddrinfo will return one entry per allowed protocol family
    // containing the loopback address for that family.
    //

    memset(&Hints, 0, sizeof(Hints));
    Hints.ai_family = Family;
    Hints.ai_socktype = SocketType;
    iResult = getaddrinfo(Server, PortName, &Hints, &AddrInfo);
    if (iResult != 0) {
        printf("Cannot resolve address [%s] and port [%s], error %d: %s\n",
               Server, PortName, WSAGetLastError(), gai_strerror(iResult));
        return SOCKET_ERROR;
    }
    return 0;
}

```

#### NOTE

The version of the [getaddrinfo](#) function that supports IPv6 is new for the Windows XP release of Windows.

#### Code to Avoid

Host address translation has traditionally been achieved using the [gethostbyname](#) function. Beginning with Windows XP:

- The [getaddrinfo](#) function makes the [gethostbyname](#) function obsolete.
- Your applications should use the [getaddrinfo](#) function instead of the [gethostbyname](#) function.

#### Coding Tasks

##### To modify an existing IPv4 application to add support for IPv6

1. Acquire the Checkv4.exe utility. This utility is installed as part of the Windows SDK. The Windows SDK is available through an MSDN subscription and can also be downloaded from the Microsoft website

(<https://msdn.microsoft.com>). An older version of the *Checkv4.exe* tool was also included as part of the Microsoft IPv6 Technology Preview for Windows 2000.

2. Run the *Checkv4.exe* utility against your code. See [Using the Checkv4.exe Utility](#) to learn about running the version utility against your files.
3. The utility alerts you to usage of the **gethostbyname**, **gethostbyaddr**, and other IPv4-only functions, and provides recommendations on how to replace them with the IPv6-compatible function such as **getaddrinfo** and **getnameinfo**.
4. Replace any instances of the **gethostbyname** function, and associated code as appropriate, with the **getaddrinfo** function. On Windows Vista, use the **WSAConnectByName** or **WSAConnectByList** function when appropriate.
5. Replace any instances of the **gethostbyaddr** function, and associated code as appropriate, with the **getnameinfo** function.

Alternatively, you can search your code base for instances of the **gethostbyname** and **gethostbyaddr** functions, and change all such usage (and other associated code, as appropriate) to the **getaddrinfo** and **getnameinfo** functions.

## Related topics

[IPv6 Guide for Windows Sockets Applications](#)

[Changing Data Structures for IPv6 Winsock Applications](#)

[Dual-Stack Sockets for IPv6 Winsock Applications](#)

[Use of Hardcoded IPv4 Addresses](#)

[User Interface Issues for IPv6 Winsock Applications](#)

[Underlying Protocols for IPv6 Winsock Applications](#)

# Use of Hardcoded IPv4 Addresses

3/5/2021 • 2 minutes to read • [Edit Online](#)

The longevity of IPv4 resulted in hard coding many well-known IPv4 addresses, such as loopback addresses (127.x.x.x), integer constants such as INADDR\_LOOPBACK, among others. The practice of hard coding these addresses presents obvious problems when modifying and existing application to support IPv6 or creating new IP version-independent programs.

## Best Practice

- The best approach is to avoid hardcoding any addresses.

## Code To Avoid

- Avoid using hardcoded addresses in code.

## To modify your existing code base from IPv4 to IPv4- and IPv6-interoperability

1. Acquire the *Checkv4.exe* utility. The *Checkv4.exe* utility is installed as part of the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later. The Windows SDK is available through an MSDN subscription and can also be downloaded from the Microsoft website (<https://msdn.microsoft.com>).
2. Run the *Checkv4.exe* utility against your code. Learn about how to run the *Checkv4.exe* utility against your files in the section on [Using the Checkv4.exe Utility](#).
3. The *Checkv4.exe* utility alerts you to the presence of common defines for IPv4 addresses, such as INADDR\_LOOPBACK. Modify any code that uses literal strings with code that is protocol-version agnostic.
4. Search your code base for other potential literal strings, as appropriate.

The *Checkv4.exe* utility can help you find common literal strings, but there may be others that are specific to your application. You should perform thorough searching and testing to ensure your code base has eradicated potential problems associated with literal strings.

## Related topics

[IPv6 Guide for Windows Sockets Applications](#)

[Changing Data Structures for IPv6 Winsock Applications](#)

[Dual-Stack Sockets for IPv6 Winsock Applications](#)

[Function Calls for IPv6 Winsock Applications](#)

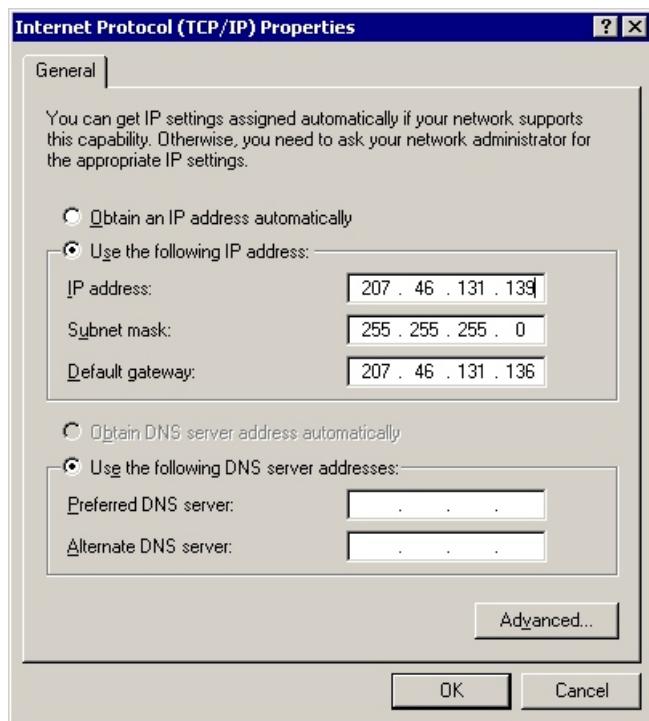
[User Interface Issues for IPv6 Winsock Applications](#)

[Underlying Protocols for IPv6 Winsock Applications](#)

# User Interface Issues for IPv6 Winsock Applications

3/5/2021 • 4 minutes to read • [Edit Online](#)

One of the most obvious changes from IPv4 to IPv6 is the size of the IP address. Many user interfaces provide dialog boxes that enable a user to enter an IP address, as exemplified in the following figure.



Addressing in IPv6, due to many factors such as length, complexity, and the significance of sections within the IPv6 address space, is not conducive to modification or specification by users. Therefore, the need to provide users with the capability of specifying their own address is reduced. Additionally, due to the complexity associated with IPv6 addressing, providing administrators with the capability of specifying IPv6 address information is not likely to occur on a per-node basis.

Displaying an IPv6 address in the UI is not inconceivable, and therefore developers should consider the variability in the size of an IPv6 address when modifying an application to support IPv6.

The rest of this section discusses the difference between IPv4 address length predictability and IPv6 address length considerations. This section presumes IPv6 addresses are being displayed in their hexadecimal representation.

IPv4 addresses are predictable in size, because they rigidly follow dotted decimal notation, as the following address example illustrates:

```
10.10.256.1
```

IPv6 addresses are not so predictable, due to the IPv6 address convention that enables the use of a double-colon (::) to represent a series of zeros. As such, the following IPv6 address representations equate to the same IPv6 address:

```
1040:0:0:0:0:0:1  
1040::1
```

The capability to represent a series of zeros with a double-colon results in an unpredictable length for any given IPv6, which requires programmers to take this capability into consideration when creating user interface displays of IPv6 addresses. Certainly, developers should ensure that the user interface is capable of displaying IP addresses that do not use a double-colon to represent a series of zeros (first address below), as well as being capable of displaying the longest possible IPv6 address (second address below, with the embedded IPv4 address) when creating their IPv6-capable user interface. Note, too, that adding the Scope identifier (ID) to the following address would increase its length by as much as another eleven characters:

```
21DA:00D3:0010:2F3B:02AA:00FF:FE28:9C5A  
0000:0000:0000:0000:ffff:123.123.123.123
```

Another important consideration is whether name-based addresses are more appropriate than number-based IPv6 addresses. If name-based addresses are more appropriate, consideration for naming conventions should be built into the user interface, including any input error-checking appropriate for the task.

There are other complexities associated with displaying IPv6 addresses that developers must take into consideration when modifying their application, and when designing user interface representations of IPv6 addresses. Some of these considerations are the following:

- Should the address contain all sequences of zeros, or use the double-colon notation?
- Is it more appropriate to use a number-based address representation or a name-based representation?
- Is the user interested in discerning a certain aspect of the addressing scheme, such as the subnet prefix, scope identifier, or other subfields?
- Is the user interested in determining other aspects of the address, such as the TLA identifier, the NLA identifier, or the SLA identifier?
- Will your user interface be capable of discerning embedded IPv6 addresses, and if so, how will those be handled and displayed? Will you discern between IPv4-compatible addresses and IPv4-mapped IPv6 addresses when displaying address information to the user?

There are other considerations as well, and developers should carefully consider their customer audience when developing IP address user interfaces.

### Best Practices

- Developers must consider the appropriate approach to each user interface when modifying their application to support IPv6. Ensuring that the user interface contains sufficient length to display IPv6 addresses is imperative, as is determining whether that address is number or name based.
- Whenever possible, use existing Winsock and IP Helper functions when using IPv6 addresses rather than re-implementing this logic. For example, the [RtlIpv6AddressToString](#), [RtlIpv6AddressToStringEx](#), [RtlIpv6StringToAddress](#), and [RtlIpv6StringToAddressEx](#) functions can be used to convert between IPv6 addresses and string representations of these IPv6 addresses.

### Code To Avoid

- User interface elements that depend on an IPv4-sized address must undergo scrutiny, and part of that scrutiny should include whether the information you were providing (under IPv4) is appropriate for IPv6.
- The capability to specify an IP address should also depend on whether IPv4 is in use, or IPv6 is available. If IPv6 is available, is it appropriate to specify number-based (hexadecimal) addresses or name-based addresses?

### Coding Tasks

#### To revise your existing code base from IPv4 to IPv4- and IPv6-interoperability

1. Perform a visual review of the user interface, looking for any element that is dependent on a specific length

for the IP address string. Controls with the easily identified four-section dotted decimal notation are easy to spot, but others are not. There may be places where IP addresses could be displayed, such as in dialog boxes, where an IPv6 address might run out of display room.

2. Upon finding any of these controls, scrutinize whether it is appropriate to display the address when using IPv6. If it is possible that either IPv4 or IPv6 is in use, ensure that the user interface can accommodate either. Replace or augment any controls with user interface controls that can display an entire IPv6 address.
3. Follow up with testing of the user interface to ensure the changes that enable IPv6 address display maintain the intended usability when using IPv4 addresses. Also, test for protocol address display locations, such as informational dialog boxes, to ensure they properly handle IPv6 addresses.

## Related topics

[IPv6 Guide for Windows Sockets Applications](#)

[Changing Data Structures for IPv6 Winsock Applications](#)

[Dual-Stack Sockets for IPv6 Winsock Applications](#)

[Function Calls for IPv6 Winsock Applications](#)

[Use of Hardcoded IPv4 Addresses](#)

[Underlying Protocols for IPv6 Winsock Applications](#)

# Underlying Protocols for IPv6 Winsock Applications

3/5/2021 • 2 minutes to read • [Edit Online](#)

Some protocols that your application depends on for certain services, such as Remote Procedure Call (RPC), may have IP-version dependent functions and structures that require you to modify your application in terms of their usage.

Part of the process to modify your code to support IPv6 should include determining whether the use of the underlying protocols (from the perspective of the stack, these protocols are considered higher-layer protocols) require modification in order to properly make use of IPv6.

## Related topics

[IPv6 Guide for Windows Sockets Applications](#)

[Changing Data Structures for IPv6 Winsock Applications](#)

[Dual-Stack Sockets for IPv6 Winsock Applications](#)

[Function Calls for IPv6 Winsock Applications](#)

[Use of Hardcoded IPv4 Addresses](#)

[User Interface Issues for IPv6 Winsock Applications](#)

# Dual-Stack Sockets for IPv6 Winsock Applications

3/5/2021 • 5 minutes to read • [Edit Online](#)

In order to support both IPv4 and IPv6 on Windows XP with Service Pack 1 (SP1) and on Windows Server 2003, an application has to create two sockets, one socket for use with IPv4 and one socket for use with IPv6. These two sockets must be handled separately by the application.

Windows Vista and later offer the ability to create a single IPv6 socket which can handle both IPv6 and IPv4 traffic. For example, a TCP listening socket for IPv6 is created, put into dual stack mode, and bound to port 5001. This dual-stack socket can accept connections from IPv6 TCP clients connecting to port 5001 and from IPv4 TCP clients connecting to port 5001. This feature allows for greatly simplified application design and reduces the resource overhead required of posting operations on two separate sockets.

## Creating a Dual-Stack Socket

By default, an IPv6 socket created on Windows Vista and later only operates over the IPv6 protocol. In order to make an IPv6 socket into a dual-stack socket, the [setsockopt](#) function must be called with the **IPV6\_V6ONLY** socket option to set this value to zero before the socket is bound to an IP address. When the **IPV6\_V6ONLY** socket option is set to zero, a socket created for the **AF\_INET6** address family can be used to send and receive packets to and from an IPv6 address or an IPv4 mapped address.

## IP Addresses with a Dual-Stack Socket

Dual-stack sockets always require IPv6 addresses. The ability to interact with an IPv4 address requires the use of the IPv4-mapped IPv6 address format. Any IPv4 addresses must be represented in the IPv4-mapped IPv6 address format which enables an IPv6 only application to communicate with an IPv4 node. The IPv4-mapped IPv6 address format allows the IPv4 address of an IPv4 node to be represented as an IPv6 address. The IPv4 address is encoded into the low-order 32 bits of the IPv6 address, and the high-order 96 bits hold the fixed prefix 0:0:0:0:FFFF. The IPv4-mapped IPv6 address format is specified in RFC 4291. For more information, see [www.ietf.org/rfc/rfc4291.txt](http://www.ietf.org/rfc/rfc4291.txt). The IN6ADDR\_SETV4MAPPED macro in *Mstcpip.h* can be used to convert an IPv4 address to the required IPv4-mapped IPv6 address format.

If the underlying protocol is actually IPv4, then the IPv4 address is mapped into an IPv4-mapped IPv6 address format. That is, the family field in the [SOCKADDR](#) structure indicates AF\_INET6, but an IPv4-mapped IPv6 address is encoded in the IPv6 address structure. For a dual-stack socket in listening mode, this means that any accepted IPv4 connections will return an IPv4-mapped IPv6 address. For a dual-stack socket that is connecting to an IPv4 destination, the SOCKADDR structure passed to connect must be an IPv4-mapped IPv6 address. Applications must take care to handle these IPv4-mapped IPv6 addresses appropriately and only use them with dual stack sockets. If an IP address is to be passed to a regular IPv4 socket, the address must be a regular IPv4 address not a IPv4-mapped IPv6 address.

## Potential Issues using a Dual-Stack Socket

A potential pitfall for applications is getting an IPv4-mapped IPv6 address on a dual-stack socket and then trying to use the returned IP address on a different IPv6 only socket. For example, the [getsockname](#) or [getpeername](#) functions can return an IPv4-mapped IPv6 address when used on a dual-stack socket. If the returned IPv4-mapped IPv6 address is then subsequently used on a different socket that was not set to dual-stack (an IPv6 only socket which is the default behavior when a socket is created), any use of this IPv6 only socket with an IPv4-mapped IPv6 address will fail. The IPv4-mapped IPv6 address format can only be used on a dual-stack socket.

On a dual-stack datagram socket, if an application requires the [LPFN\\_WSARECVMMSG \(WSARecvMsg\)](#) function to return packet information in a [WSAMSG](#) structure for datagrams received over IPv4, then [IP\\_PKTINFO](#) socket option must be set to true on the socket. If only the [IPV6\\_PKTINFO](#) option is set to true on the socket, packet information will be provided for datagrams received over IPv6 but may not be provided for datagrams received over IPv4.

If an application tries to set the [IP\\_PKTINFO](#) socket option on a dual-stack datagram socket and IPv4 is disabled on the system, then the [setsockopt](#) function will fail and [WSAGetLastError](#) will return with an error of [WSAEINVAL](#). This same error is also returned by the [setsockopt](#) function as a result of other errors. If an application tries to set an [IPPROTO\\_IP](#) level socket option on a dual-stack socket and it fails with [WSAEINVAL](#), then the application should determine if IPv4 is disabled on the local computer. One method that can be used to detect if IPv4 is enabled or disabled is to call the [socket](#) function with the *af* parameter set to AF\_INET to try and create an IPv4 socket. If the [socket](#) function fails and [WSAGetLastError](#) returns an error of [WSAEAFNOSUPPORT](#), then it means IPv4 is not enabled. In this case, a [setsockopt](#) function failure when attempting to set the [IP\\_PKTINFO](#) socket option can be ignored by the application. Otherwise a failure when attempting to set the [IP\\_PKTINFO](#) socket option should be treated as an unexpected error.

For a dual-stack socket when sending datagrams with the [WSASendMsg](#) function and an application wants to specify a specific local IP source address to be used, the method to handle this depends on the destination IP address. When sending to an IPv4 destination address or an IPv4-mapped IPv6 destination address, one of the control data objects passed in the [WSAMSG](#) structure pointed to by the *lpMsg* parameter should contain an [in\\_pktinfo](#) structure containing the local IPv4 source address to use for sending. When sending to an IPv6 destination address that is not a an IPv4-mapped IPv6 address, one of the control data objects passed in the [WSAMSG](#) structure pointed to by the *lpMsg* parameter should contain an [in6\\_pktinfo](#) structure containing the local IPv6 source address to use for sending.

## Related topics

[IPv6 Guide for Windows Sockets Applications](#)

[Changing Data Structures for IPv6 Winsock Applications](#)

[Function Calls for IPv6 Winsock Applications](#)

[Use of Hardcoded IPv4 Addresses](#)

[User Interface Issues for IPv6 Winsock Applications](#)

[Underlying Protocols for IPv6 Winsock Applications](#)

[getpeername](#)

[getsockname](#)

[in\\_pktinfo](#)

[in6\\_pktinfo](#)

[IP\\_PKTINFO](#)

[IPV6\\_PKTINFO](#)

[setsockopt](#)

[LPFN\\_WSARECVMMSG \(WSARecvMsg\)](#)

[WSASendMsg](#)



# Using the Checkv4.exe utility

3/5/2021 • 3 minutes to read • [Edit Online](#)

## IMPORTANT

The *Checkv4.exe* utility doesn't ship in the Windows Software Development Kit (SDK) for Windows 8, nor in later versions of the Windows SDK.

The *Checkv4.exe* utility is designed to provide you with a code porting partner; a utility that steps through your code base with you, identifies potential problems or highlights code that could benefit from IPv6-capable functions or structures, and makes recommendations. With the *Checkv4.exe* utility, the task of modifying an existing IPv4 application to support IPv6 becomes much easier.

The *Checkv4.exe* utility is installed as part of the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later SDKs (up to, but not including, the Windows Software Development Kit (SDK) for Windows 8).

An earlier version of the *Checkv4.exe* utility with more limited features was also made available as part of the earlier Microsoft IPv6 Technology Preview for Windows 2000.

The following sections describe how to use the *Checkv4.exe* utility, then explain the recommended approach for modifying an existing IPv4 application to support IPv6.

## Recommendations for Running Checkv4.exe

- The *Checkv4.exe* utility is straightforward. Simply execute *Checkv4.exe* at the command line with the name of the file you want to check as the parameter. *Checkv4.exe* parses the file and provides feedback as to where IPv6 porting issues exist in that file. Placing the *Checkv4.exe* into your computer's path makes running the *Checkv4.exe* utility from anywhere in your source code directory structure much easier. For example, placing *Checkv4.exe* into %windir% enables you to launch *Checkv4.exe* from any directory on your computer without including its path.
- Issue the following command at the command prompt to parse the file Simplec.c:

**Checkv4 simplec.c**

Note that some of the recommendations made by the *Checkv4.exe* utility require structures available only in recent versions of the *Ws2tcpip.h* header file, such as the **SOCKADDR\_IN6** structure. These header files are included in the Windows SDK released for Windows Vista and later. These header files are also included in the earlier Platform Software Development Kit (SDK) released for Windows Server 2003. These header files are also included as part of an MSDN subscription or by download.

The following screen shot displays the results of using the *Checkv4.exe* utility on the Simplec.c file included in Appendix A:

```
D:\MS\Admin\Technologies\WS\PortingCode\IPv4>checkv4 simplec.c
simplec.c<45> : sockaddr_in : use sockaddr_storage instead, or use soc
simplec.c<101> : gethostbyname : use getaddrinfo instead
simplec.c<102> : gethostbyaddr : use getnameinfo instead
simplec.c<105> : gethostbyname : use getaddrinfo instead
simplec.c<108> : inet_addr : use WSASTringToAddress or getaddrinfo wit
simplec.c<109> : AF_INET : use AF_INET6 in addition for IPv6 support
simplec.c<109> : gethostbyaddr : use getnameinfo instead
simplec.c<119> : sockaddr_in : use sockaddr_storage instead, or use so
simplec.c<126> : AF_INET : use AF_INET6 in addition for IPv6 support

D:\MS\Admin\Technologies\WS\PortingCode\IPv4>
```

The following screen shot displays the results of using the *Checkv4.exe* utility on the Simplec.c file, which is also included in Appendix A:

```
D:\MS\Admin\Technologies\WS\PortingCode\IPv4>checkv4 simplec.c
simplec.c<28> : INADDR_ANY : use getaddrinfo with nodename=NULL and AI_PA
simplec.c<41> : sockaddr_in : use sockaddr_storage instead, or use sockad
simplec.c<86> : AF_INET : use AF_INET6 in addition for IPv6 support
simplec.c<87> : INADDR_ANY : use getaddrinfo with nodename=NULL and AI_PA
simplec.c<87> : inet_addr : use WSASTringToAddress or getaddrinfo with AI
simplec.c<94> : AF_INET : use AF_INET6 in addition for IPv6 support
simplec.c<144> : inet_ntoa : use WSAAStringToString or getnameinfo with N
simplec.c<163> : inet_ntoa : use WSAAAddressToString or getnameinfo with N

D:\MS\Admin\Technologies\WS\PortingCode\IPv4>
```

## The Application Modification Process: Where to Start

There is a recommended procedure associated with adding IPv6 capability to applications. Following this sequence is beneficial, because it enables developers to ensure that all steps necessary to modify an existing IPv4 application to support IPv6 are taken. Certain applications may require more extensive attention to one of these sequences; for example, a system service would likely have less user interface issues than a graphical file transfer program (FTP).

### To modify IPv4 applications to support IPv6

1. Fix structures and declarations to enable IPv6 and IPv4 compatibility.
2. Modify function calls to take advantage of IPv6-enabled functions, such as the [getaddrinfo](#) and [getnameinfo](#) functions.
3. Review source code for the use of hard-coded IPv4 addresses such as the loopback address, or the use of other literal strings.
4. Perform a thorough review of the user interface, including informational dialog boxes. Give thought to whether it is appropriate for IPv6-enabled applications to specify or provide IP-address based information.
5. Determine whether your application relies on underlying protocols, such as RPC, and make appropriate programmatic changes to handle IPv6 addresses.
6. Use the compile-time flag IPV6STRICT when compiling applications on Windows XP and later. This flag results in incompatible code failing to compile, as follows:

Windows Sockets 1.x applications with incompatible code fail to compile and return the error message "WINSOCK2 Required."

Windows Sockets 2.x applications with incompatible code cause a compile time error for each instance of incompatible code. An error message is generated in the following format:

```
[file name] ([line number]) : [error message] '[symbol]_IPV6INCOMPATIBLE'
```

For example:

```
sample.c(8) : error C2065: 'gethostbyaddr_IPV6INCOMPATIBLE' : undeclared identifier
```

# Appendix A: IPv4-only Source Code

3/5/2021 • 2 minutes to read • [Edit Online](#)

This appendix provides the source code for the Platform Software Development Kit (SDK) samples Simplec.c and Simples.c, which are used throughout this IPv6 guide as examples:

- [IPv4-only Client Code](#)
- [IPv4-only Server Code](#)

## WARNING

This code is included in this guide to contrast the IP-version agnostic code provided in Appendix B. This sample code is provided for comparison purposes, and to provide a clear and readily available reference while reading this IPv6 guide. An IP version agnostic version of this source code is provided in [Appendix B: IP-version Agnostic Source Code](#).

## NOTE

By comparing the source code in Appendix A (IPv4 only) and Appendix B (IP version agnostic), you get a sense of the changes necessary to modify your Windows Sockets application to add support for IPv6.

# IPv4-only Client Code

3/5/2021 • 3 minutes to read • [Edit Online](#)

The following code is the Simplec.c file, which is an IPv4-only Windows Sockets client (an IPv6 enabled version of the Simplec.c file can be found in Appendix B). This code is provided for comparison purposes only — use Appendix B for an example of how to write an IPv6-enabled client.

```
/*****************************************************************************\
* simplec.c - Simple TCP/UDP client using Winsock 1.1
*
*      This is a part of the Microsoft<entity type="reg"/> Source Code Samples.
*      Copyright 1996 - 2000 Microsoft Corporation.
*      All rights reserved.
*      This source code is only intended as a supplement to
*      Microsoft Development Tools and/or WinHelp<entity type="reg"/> documentation.
*      See these sources for detailed information regarding the
*      Microsoft samples programs.
\****************************************************************************/
#ifndef UNICODE
#define UNICODE
#endif

#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

#define DEFAULT_PORT 5001
#define DEFAULT_PROTO SOCK_STREAM // TCP

void Usage(char *progname) {
    fprintf(stderr,"Usage\n%s -p [protocol] -n [server] -e [endpoint] \
    -l [iterations]\n",
    progname);
    fprintf(stderr,"Where:\n\tprotocol is one of TCP or UDP\n");
    fprintf(stderr,"\\tserver is the IP address or name of server\n");
    fprintf(stderr,"\\tendpoint is the port to listen on\n");
    fprintf(stderr,"\\titerations is the number of loops to execute\n");
    fprintf(stderr,"\\t(-1 by itself makes client run in an infinite loop,);
    fprintf(stderr," Hit Ctrl-C to terminate it)\n");
    fprintf(stderr,"Defaults are TCP , localhost and 5001\n");
    WSACleanup();
    exit(1);
}

int main(int argc, char **argv) {

    char Buffer[128];
    char *server_name= "localhost";
    unsigned short port = DEFAULT_PORT;
    int retval, loopflag=0;
    int i, loopcount,maxloop=-1;
    unsigned int addr;
    int socket_type = DEFAULT_PROTO;
    struct sockaddr_in server;
    struct hostent *hp;
    WSADATA wsaData;
```

```

WSADATA wsaData;
SOCKET conn_socket;

if (argc >1) {
    for(i=1;i <argc;i++) {
        if ( (argv[i][0] == '-') || (argv[i][0] == '/') ) {
            switch(tolower(argv[i][1])) {
                case 'p':
                    if (!_stricmp(argv[i+1], "TCP") )
                        socket_type = SOCK_STREAM;
                    else if (!_stricmp(argv[i+1], "UDP") )
                        socket_type = SOCK_DGRAM;
                    else
                        Usage(argv[0]);
                    i++;
                    break;

                case 'n':
                    server_name = argv[++i];
                    break;
                case 'e':
                    port = (USHORT) atoi(argv[++i]);
                    break;
                case 'l':
                    loopflag =1;
                    if (argv[i+1]) {
                        if (argv[i+1][0] != '-')
                            maxloop = atoi(argv[i+1]);
                        }
                    else
                        maxloop = -1;
                    i++;
                    break;
                default:
                    Usage(argv[0]);
                    break;
            }
        }
        else
            Usage(argv[0]);
    }
}

if ((retval = WSAStartup(0x202,&wsaData)) != 0) {
    fprintf(stderr,"WSAStartup failed with error %d\n",retval);
    WSACleanup();
    return -1;
}

if (port == 0){
    Usage(argv[0]);
}

// Attempt to detect if we should call gethostbyname() or
// gethostbyaddr()

if (isalpha(server_name[0])) { /* server address is a name */
    hp = gethostbyname(server_name);
}
else { /* Convert nnn.nnn address to a usable one */
    addr = inet_addr(server_name);
    hp = gethostbyaddr((char *)&addr,4,AF_INET);
}
if (hp == NULL ) {
    fprintf(stderr,"Client: Cannot resolve address [%s]: Error %d\n",
           server_name,WSAGetLastError());
    WSACleanup();
    exit(1);
}

```

```

}

// 
// Copy the resolved information into the sockaddr_in structure
//
memset(&server,0,sizeof(server));
memcpy(&(server.sin_addr),hp->h_addr,hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = htons(port);

conn_socket = socket(AF_INET,socket_type,0); /* Open a socket */
if (conn_socket <0 ) {
    fprintf(stderr,"Client: Error Opening socket: Error %d\n",
        WSAGetLastError());
    WSACleanup();
    return -1;
}

// 
// Notice that nothing in this code is specific to whether we
// are using UDP or TCP.
// We achieve this by using a simple trick.
//     When connect() is called on a datagram socket, it does not
// actually establish the connection as a stream (TCP) socket
// would. Instead, TCP/IP establishes the remote half of the
// ( LocalIPAddress, LocalPort, RemoteIP, RemotePort) mapping.
// This enables us to use send() and recv() on datagram sockets,
// instead of recvfrom() and sendto()

printf("Client connecting to: %s\n",hp->h_name);
if (connect(conn_socket,(struct sockaddr*)&server,sizeof(server))
== SOCKET_ERROR) {
    fprintf(stderr,"connect() failed: %d\n",WSAGetLastError());
    WSACleanup();
    return -1;
}

// cook up a string to send
//
loopcount =0;
while(1) {
    sprintf_s(Buffer,sizeof(Buffer), "This is a small test message [number %d]",loopcount++);
    retval = send(conn_socket,Buffer,sizeof(Buffer),0);
    if (retval == SOCKET_ERROR) {
        fprintf(stderr,"send() failed: error %d\n",WSAGetLastError());
        WSACleanup();
        return -1;
    }
    printf("Sent Data [%s]\n",Buffer);
    retval = recv(conn_socket,Buffer,sizeof (Buffer),0 );
    if (retval == SOCKET_ERROR) {
        fprintf(stderr,"recv() failed: error %d\n",WSAGetLastError());
        closesocket(conn_socket);
        WSACleanup();
        return -1;
    }
}
// 
// We are not likely to see this with UDP, since there is no
// 'connection' established.
//
if (retval == 0) {
    printf("Server closed connection\n");
    closesocket(conn_socket);
    WSACleanup();
    return -1;
}
printf("Received %d bytes, data [%s] from server\n",retval,Buffer);
if (!loopflag){

```

```
    printf("Terminating connection\n");
    break;
}
else {
    if ( (loopcount >= maxloop) && (maxloop >0) )
        break;
}
closesocket(conn_socket);
WSACleanup();
}
```

# IPv4-only Server Code

3/5/2021 • 3 minutes to read • [Edit Online](#)

The following code is the Simples.c file, which is an IPv4-only Windows Sockets server (an IPv6-enabled version of the Simples.c file can be found in Appendix B). This code is provided for comparison purposes only — use Appendix B for an example of how to write an IPv6-enabled server.

```
*****\n/* simples.c - Simple TCP/UDP server using Winsock 1.1\n*      This is a part of the Microsoft<entity type="reg"/> Source Code Samples.\n*      Copyright 1996 - 2000 Microsoft Corporation.\n*      All rights reserved.\n*      This source code is only intended as a supplement to\n*      Microsoft Development Tools and/or WinHelp documentation.\n*      See these sources for detailed information regarding the\n*      Microsoft samples programs.\n*****/\n#define WIN32_LEAN_AND_MEAN\n\n#include <winsock2.h>\n#include <Ws2tcpip.h>\n#include <stdio.h>\n#include <stdlib.h>\n\n// Link with ws2_32.lib\n#pragma comment(lib, "Ws2_32.lib")\n\n#define STRICMP _stricmp\n\n#define DEFAULT_PORT 5001\n#define DEFAULT_PROTO SOCK_STREAM // TCP\n\n\nvoid Usage(char *progname) {\n    fprintf(stderr,"Usage\n%s -p [protocol] -e [endpoint] -i [interface]\n",\n           progname);\n    fprintf(stderr,"Where:\n\tprotocol is one of TCP or UDP\n");\n    fprintf(stderr,"\\tendpoint is the port to listen on\n");\n    fprintf(stderr,"\\tinterface is the ipaddr (in dotted decimal notation)\n");\n    fprintf(stderr," to bind to\n");\n    fprintf(stderr,"Defaults are TCP,5001 and INADDR_ANY\n");\n    WSACleanup();\n    exit(1);\n}\nint main(int argc, char **argv) {\n\n    char Buffer[128];\n    char *interface= NULL;\n    unsigned short port=DEFAULT_PORT;\n    int retval;\n    int fromlen;\n    int i;\n    int socket_type = DEFAULT_PROTO;\n    struct sockaddr_in local, from;\n    WSADATA wsaData;\n    SOCKET listen_socket, msgsock;\n\n    /* Parse arguments */\n    if (argc >1) {\n        for(i=1;i <argc;i++) {\n            if ((argv[i][0] == '-' || argv[i][0] == '/') &&
```

```

        switch(tolower(argv[i][1])) {
            case 'p':
                if (!STRICMP(argv[i+1], "TCP"))
                    socket_type = SOCK_STREAM;
                else if (!STRICMP(argv[i+1], "UDP"))
                    socket_type = SOCK_DGRAM;
                else
                    Usage(argv[0]);
                i++;
                break;

            case 'i':
                interface = argv[++i];
                break;
            case 'e':
                port = (unsigned short) atoi(argv[++i]);
                break;
            default:
                Usage(argv[0]);
                break;
        }
    }
    else
        Usage(argv[0]);
}

if ((retval = WSASStartup(0x202,&wsaData)) != 0) {
    fprintf(stderr,"WSASStartup failed with error %d\n",retval);
    WSACleanup();
    return -1;
}

if (port == 0){
    Usage(argv[0]);
}

local.sin_family = AF_INET;
local.sin_addr.s_addr = (!interface)?INADDR_ANY:inet_addr(interface);

/*
 * Port MUST be in Network Byte Order
 */
local.sin_port = htons(port);

listen_socket = socket(AF_INET, socket_type,0); // TCP socket

if (listen_socket == INVALID_SOCKET){
    fprintf(stderr,"socket() failed with error %d\n",WSAGetLastError());
    WSACleanup();
    return -1;
}
// bind() associates a local address and port combination with the
// socket just created. This is most useful when the application is a
// server that has a well-known port that clients know about in advance.
//

if (bind(listen_socket,(struct sockaddr*)&local,sizeof(local) )
    == SOCKET_ERROR) {
    fprintf(stderr,"bind() failed with error %d\n",WSAGetLastError());
    WSACleanup();
    return -1;
}

//
// So far, everything we did was applicable to TCP as well as UDP.
// However, there are certain steps that do not work when the server is
// using UDP.

```

```

// using UDP.
//

// We cannot listen() on a UDP socket.

if (socket_type != SOCK_DGRAM) {
    if (listen(listen_socket,5) == SOCKET_ERROR) {
        fprintf(stderr,"listen() failed with error %d\n",WSAGetLastError());
        WSACleanup();
        return -1;
    }
}
printf("%s: 'Listening' on port %d, protocol %s\n",argv[0],port,
       (socket_type == SOCK_STREAM)? "TCP": "UDP");
while(1) {
    fromlen = sizeof(from);
    //
    // accept() doesn't make sense on UDP, since we do not listen()
    //
    if (socket_type != SOCK_DGRAM) {
        msgsock = accept(listen_socket,(struct sockaddr*)&from, &fromlen);
        if (msgsock == INVALID_SOCKET) {
            fprintf(stderr,"accept() error %d\n",WSAGetLastError());
            WSACleanup();
            return -1;
        }
        printf("accepted connection from %s, port %d\n",
               inet_ntoa(from.sin_addr),
               htons(from.sin_port)) ;
    }
    else
        msgsock = listen_socket;

    //
    // In the case of SOCK_STREAM, the server can do recv() and
    // send() on the accepted socket and then close it.

    // However, for SOCK_DGRAM (UDP), the server will do
    // recvfrom() and sendto() in a loop.

    if (socket_type != SOCK_DGRAM)
        retval = recv(msgsock,Buffer,sizeof (Buffer),0 );
    else {
        retval = recvfrom(msgsock,Buffer,sizeof (Buffer),0,
                          (struct sockaddr *)&from,&fromlen);
        printf("Received datagram from %s\n",inet_ntoa(from.sin_addr));
    }

    if (retval == SOCKET_ERROR) {
        fprintf(stderr,"recv() failed: error %d\n",WSAGetLastError());
        closesocket(msgsock);
        continue;
    }
    if (retval == 0) {
        printf("Client closed connection\n");
        closesocket(msgsock);
        continue;
    }
    printf("Received %d bytes, data [%s] from client\n",retval,Buffer);

    printf("Echoing same data back to client\n");
    if (socket_type != SOCK_DGRAM)
        retval = send(msgsock,Buffer,sizeof(Buffer),0);
    else
        retval = sendto(msgsock,Buffer,sizeof (Buffer),0,
                       (struct sockaddr *)&from,fromlen);
    if (retval == SOCKET_ERROR) {
        fprintf(stderr,"send() failed: error %d\n",WSAGetLastError());

```

```
    }
    if (socket_type != SOCK_DGRAM){
        printf("Terminating connection\n");
        closesocket(msgsock);
    }
    else
        printf("UDP server looping back for more requests\n");
    continue;
}
}
```

# Appendix B: IP-version Agnostic Source Code

3/5/2021 • 2 minutes to read • [Edit Online](#)

This appendix illustrates a rewritten version of the Simplec.c and Simples.c sample applications that gracefully handles either IPv4 or IPv6.

- [IPv6-Enabled Client Code](#)
- [IPv6-Enabled Server Code](#)

This code exemplifies the guidelines set forth in this IPv6 guide, and is included to provide source code that has been successfully modified to add support for IPv6. This sample is intentionally simple, but provides a hands-on sample for perusing and reviewing. An IPv4 only version of this source code is provided in [Appendix A: IPv4-only Source Code](#).

By comparing the source code in Appendix A (IPv4 only) and Appendix B (IP-version agnostic), you get a sense of the changes necessary to modify your Windows Sockets application to add support for IPv6.

# IPv6-Enabled Client Code

3/5/2021 • 7 minutes to read • [Edit Online](#)

The following code is the IP agnostic Client.c file, which is an IPv6-enabled version of the Simplec.c file.

```
// client.c - Simple TCP/UDP client using Winsock 2.2
//
//      This is a part of the Microsoft<entity type="reg"/> Source Code Samples.
//      Copyright 1996 - 2000 Microsoft Corporation.
//      All rights reserved.
//      This source code is only intended as a supplement to
//      Microsoft Development Tools and/or WinHelp<entity type="reg"/> documentation.
//      See these sources for detailed information regarding the
//      Microsoft samples programs.
//
#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>
//#include <string.h>

// Needed for the Windows 2000 IPv6 Tech Preview.
#if (_WIN32_WINNT == 0x0500)
#include <tpipv6.h>
#endif

// Link with ws2_32.lib
#pragma comment(lib, "Ws2_32.lib")

#define STRICMP _stricmp

//
// This code assumes that at the transport level, the system only supports
// one stream protocol (TCP) and one datagram protocol (UDP). Therefore,
// specifying a socket type of SOCK_STREAM is equivalent to specifying TCP
// and specifying a socket type of SOCK_DGRAM is equivalent to specifying UDP.
//

#define DEFAULT_SERVER      NULL // Will use the loopback interface
#define DEFAULT_FAMILY       PF_UNSPEC // Accept either IPv4 or IPv6
#define DEFAULT_SOCKTYPE     SOCK_STREAM // TCP
#define DEFAULT_PORT          "5001" // Arbitrary, albeit a historical test port
#define DEFAULT_EXTRA         0 // Number of "extra" bytes to send

#define BUFFER_SIZE           65536

#define UNKNOWN_NAME "<unknown>"

void Usage(char *ProgName)
{
    fprintf(stderr, "\nSimple socket sample client program.\n");
    fprintf(stderr,
            "\n%s [-s server] [-f family] [-t transport] [-p port] [-b bytes] [-n number]\n\n",
            ProgName);
    fprintf(stderr, "    server\tServer name or IP address. (default: %s)\n",
            (DEFAULT_SERVER == NULL) ? "loopback address" : DEFAULT_SERVER);
    fprintf(stderr,
            "    family\tOne of PF_INET, PF_INET6 or PF_UNSPEC. (default: %s)\n",
            "
```

```

(DEFAULT_FAMILY ==
PF_UNSPEC) ? "PF_UNSPEC" : ((DEFAULT_FAMILY ==
PF_INET) ? "PF_INET" : "PF_INET6"));
fprintf(stderr, " transport\tEither TCP or UDP. (default: %s)\n",
(DEFAULT_SOCKTYPE == SOCK_STREAM) ? "TCP" : "UDP");
fprintf(stderr, " port\t\tPort on which to connect. (default: %s)\n",
DEFAULT_PORT);
fprintf(stderr, " bytes\t\tBytes of extra data to send. (default: %d)\n",
DEFAULT_EXTRA);
fprintf(stderr, " number\tNumber of sends to perform. (default: 1)\n");
fprintf(stderr, " (-n by itself makes client run in an infinite loop,)");
fprintf(stderr, " Hit Ctrl-C to terminate)\n");
WSACleanup();
exit(1);
}

LPTSTR PrintError(int ErrorCode)
{
    static TCHAR Message[1024];

    // If this program was multithreaded, we'd want to use
    // FORMAT_MESSAGE_ALLOCATE_BUFFER instead of a static buffer here.
    // (And of course, free the buffer when we were done with it)

    FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS |
        FORMAT_MESSAGE_MAX_WIDTH_MASK,
        NULL, ErrorCode, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        Message, 1024, NULL);
    return Message;
}

int ReceiveAndPrint(SOCKET ConnSocket, char *Buffer, int BufLen)
{
    int AmountRead;

    AmountRead = recv(ConnSocket, Buffer, BufLen, 0);
    if (AmountRead == SOCKET_ERROR) {
        fprintf(stderr, "recv() failed with error %d: %s\n",
            WSAGetLastError(), PrintError(WSAGetLastError()));
        closesocket(ConnSocket);
        WSACleanup();
        exit(1);
    }
    //
    // We are not likely to see this with UDP, since there is no
    // 'connection' established.
    //
    if (AmountRead == 0) {
        printf("Server closed connection\n");
        closesocket(ConnSocket);
        WSACleanup();
        exit(0);
    }

    printf("Received %d bytes from server: [%.*s]\n",
        AmountRead, AmountRead, Buffer);

    return AmountRead;
}

int main(int argc, char **argv)
{
    char Buffer[BUFFER_SIZE], AddrName[NI_MAXHOST];

    char *Server = DEFAULT_SERVER;
    int Family = DEFAULT_FAMILY;
    int SocketType = DEFAULT_SOCKTYPE;
    char *Port = DEFAULT_PORT;
}

```

```

WSADATA wsaData;

int i, RetVal, AddrLen, AmountToSend;
int ExtraBytes = DEFAULT_EXTRA;
unsigned int Iteration, MaxIterations = 1;
BOOL RunForever = FALSE;

ADDRINFO Hints, *AddrInfo, *AI;
SOCKET ConnSocket = INVALID_SOCKET;
struct sockaddr_storage Addr;

if (argc > 1) {
    for (i = 1; i < argc; i++) {
        if (((argv[i][0] == '-') || (argv[i][0] == '/')) &&
            (argv[i][1] != 0) && (argv[i][2] == 0)) {
            switch (tolower(argv[i][1])) {
                case 'f':
                    if (!argv[i + 1])
                        Usage(argv[0]);
                    if (!STRICMP(argv[i + 1], "PF_INET"))
                        Family = PF_INET;
                    else if (!STRICMP(argv[i + 1], "AF_INET"))
                        Family = PF_INET;
                    else if (!STRICMP(argv[i + 1], "PF_INET6"))
                        Family = PF_INET6;
                    else if (!STRICMP(argv[i + 1], "AF_INET6"))
                        Family = PF_INET6;
                    else if (!STRICMP(argv[i + 1], "PF_UNSPEC"))
                        Family = PF_UNSPEC;
                    else if (!STRICMP(argv[i + 1], "AF_UNSPEC"))
                        Family = PF_UNSPEC;
                    else
                        Usage(argv[0]);
                    i++;
                    break;
                case 't':
                    if (!argv[i + 1])
                        Usage(argv[0]);
                    if (!STRICMP(argv[i + 1], "TCP"))
                        SocketType = SOCK_STREAM;
                    else if (!STRICMP(argv[i + 1], "UDP"))
                        SocketType = SOCK_DGRAM;
                    else
                        Usage(argv[0]);
                    i++;
                    break;
                case 's':
                    if (argv[i + 1]) {
                        if (argv[i + 1][0] != '-')
                            Server = argv[++i];
                        break;
                    }
                }
            Usage(argv[0]);
            break;
        }
        case 'p':
            if (argv[i + 1]) {
                if (argv[i + 1][0] != '-')
                    Port = argv[++i];
                break;
            }
        }
    Usage(argv[0]);
    break;
}

```

```

        case 'b':
            if (argv[i + 1]) {
                if (argv[i + 1][0] != '-') {
                    ExtraBytes = atoi(argv[+i]);
                    if (ExtraBytes >
                        sizeof (Buffer) -
                        sizeof ("Message #4294967295"))
                        Usage(argv[0]);
                    break;
                }
            }
            Usage(argv[0]);
            break;

        case 'n':
            if (argv[i + 1]) {
                if (argv[i + 1][0] != '-') {
                    MaxIterations = atoi(argv[+i]);
                    break;
                }
            }
            RunForever = TRUE;
            break;

        default:
            Usage(argv[0]);
            break;
        }
    } else
        Usage(argv[0]);
}
}

// Ask for Winsock version 2.2.
if ((RetVal = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0) {
    fprintf(stderr, "WSAStartup failed with error %d: %s\n",
            RetVal, PrintError(RetVal));
    WSACleanup();
    return -1;
}
//
// By not setting the AI_PASSIVE flag in the hints to getaddrinfo, we're
// indicating that we intend to use the resulting address(es) to connect
// to a service. This means that when the Server parameter is NULL,
// getaddrinfo will return one entry per allowed protocol family
// containing the loopback address for that family.
//

memset(&Hints, 0, sizeof (Hints));
Hints.ai_family = Family;
Hints.ai_socktype = SocketType;
RetVal = getaddrinfo(Server, Port, &Hints, &AddrInfo);
if (RetVal != 0) {
    fprintf(stderr,
            "Cannot resolve address [%s] and port [%s], error %d: %s\n",
            Server, Port, RetVal, gai_strerror(RetVal));
    WSACleanup();
    return -1;
}
//
// Try each address getaddrinfo returned, until we find one to which
// we can successfully connect.
//
for (AI = AddrInfo; AI != NULL; AI = AI->ai_next) {

    // Open a socket with the correct address family for this address.

    ConnSocket = socket(AI->ai_family, AI->ai_socktype, AI->ai_protocol);

    // **** DEBUG

```

```

    // ****
    printf("socket call with family: %d socktype: %d, protocol: %d\n",
           AI->ai_family, AI->ai_socktype, AI->ai_protocol);
    if (ConnSocket == INVALID_SOCKET)
        printf("socket call failed with %d\n", WSAGetLastError());
//**** DEBUG END

    if (ConnSocket == INVALID_SOCKET) {
        fprintf(stderr, "Error Opening socket, error %d: %s\n",
                WSAGetLastError(), PrintError(WSAGetLastError()));
        continue;
    }
    //
    // Notice that nothing in this code is specific to whether we
    // are using UDP or TCP.
    //
    // When connect() is called on a datagram socket, it does not
    // actually establish the connection as a stream (TCP) socket
    // would. Instead, TCP/IP establishes the remote half of the
    // (LocalIPAddress, LocalPort, RemoteIP, RemotePort) mapping.
    // This enables us to use send() and recv() on datagram sockets,
    // instead of recvfrom() and sendto().
    //

    printf("Attempting to connect to: %s\n", Server ? Server : "localhost");
    if (connect(ConnSocket, AI->ai_addr, (int) AI->ai_addrlen) != SOCKET_ERROR)
        break;

    i = WSAGetLastError();
    if (getnameinfo(AI->ai_addr, (int) AI->ai_addrlen, AddrName,
                   sizeof(AddrName), NULL, 0, NI_NUMERICHOST) != 0)
        strcpy_s(AddrName, sizeof(AddrName), UNKNOWN_NAME);
    fprintf(stderr, "connect() to %s failed with error %d: %s\n",
            AddrName, i, PrintError(i));
    closesocket(ConnSocket);
}

if (AI == NULL) {
    fprintf(stderr, "Fatal error: unable to connect to the server.\n");
    WSACleanup();
    return -1;
}
//
// This demonstrates how to determine to where a socket is connected.
//
AddrLen = sizeof(Addr);
if (getpeername(ConnSocket, (LPSOCKADDR) &Addr, (int *) &AddrLen) == SOCKET_ERROR) {
    fprintf(stderr, "getpeername() failed with error %d: %s\n",
            WSAGetLastError(), PrintError(WSAGetLastError()));
} else {
    if (getnameinfo((LPSOCKADDR) &Addr, AddrLen, AddrName,
                   sizeof(AddrName), NULL, 0, NI_NUMERICHOST) != 0)
        strcpy_s(AddrName, sizeof(AddrName), UNKNOWN_NAME);
    printf("Connected to %s, port %d, protocol %s, protocol family %s\n",
           AddrName, ntohs(SS_PORT(&Addr)),
           (AI->ai_socktype == SOCK_STREAM) ? "TCP" : "UDP",
           (AI->ai_family == PF_INET) ? "PF_INET" : "PF_INET6");
}

// We are done with the address info chain, so we can free it.
freeaddrinfo(AddrInfo);

//
// Find out what local address and port the system picked for us.
//
AddrLen = sizeof(Addr);
if (getsockname(ConnSocket, (LPSOCKADDR) &Addr, &AddrLen) == SOCKET_ERROR) {
    fprintf(stderr, "getsockname() failed with error %d: %s\n",
            WSAGetLastError(), PrintError(WSAGetLastError()));
} else {

```

```

    if (getnameinfo((LPSOCKADDR) & Addr, AddrLen, AddrName,
                    sizeof (AddrName), NULL, 0, NI_NUMERICHOST) != 0)
        strcpy_s(AddrName, sizeof (AddrName), UNKNOWN_NAME);
    printf("Using local address %s, port %d\n",
           AddrName, ntohs(SS_PORT(&Addr)));
}

//  

// Send and receive in a loop for the requested number of iterations.  

//  

for (Iteration = 0; RunForever || Iteration < MaxIterations; Iteration++) {  

    // Compose a message to send.  

    AmountToSend =  

        sprintf_s(Buffer, sizeof (Buffer), "Message #%u", Iteration + 1);  

    for (i = 0; i < ExtraBytes; i++) {  

        Buffer[AmountToSend++] = (char) ((i & 0x3f) + 0x20);  

    }  

    // Send the message. Since we are using a blocking socket, this  

    // call shouldn't return until it's able to send the entire amount.  

    RetVal = send(ConnSocket, Buffer, AmountToSend, 0);  

    if (RetVal == SOCKET_ERROR) {  

        fprintf(stderr, "send() failed with error %d: %s\n",
               WSAGetLastError(), PrintError(WSAGetLastError()));  

        WSACleanup();  

        return -1;  

    }  

    printf("Sent %d bytes (out of %d bytes) of data: [%.*s]\n",
           RetVal, AmountToSend, AmountToSend, Buffer);  

    // Clear buffer just to prove we're really receiving something.  

    memset(Buffer, 0, sizeof (Buffer));  

    // Receive and print server's reply.  

    ReceiveAndPrint(ConnSocket, Buffer, sizeof (Buffer));  

}  

// Tell system we're done sending.  

printf("Done sending\n");
shutdown(ConnSocket, SD_SEND);  

//  

// Since TCP does not preserve message boundaries, there may still  

// be more data arriving from the server. So we continue to receive  

// data until the server closes the connection.  

//  

if (SocketType == SOCK_STREAM)
    while (ReceiveAndPrint(ConnSocket, Buffer, sizeof (Buffer)) != 0) ;  

closesocket(ConnSocket);
WSACleanup();
return 0;
}

```

# IPv6-Enabled Server Code

3/5/2021 • 8 minutes to read • [Edit Online](#)

The following code is the IP-agnostic Server.c file, which is an IPv6-enabled version of the Simple.c file.

```
//  
// server.c - Simple TCP/UDP server using Winsock 2.2  
//  
//      This is a part of the Microsoft<entity type="reg"/> Source Code Samples.  
//      Copyright 1996 - 2000 Microsoft Corporation.  
//      All rights reserved.  
//      This source code is only intended as a supplement to  
//      Microsoft Development Tools and/or WinHelp<entity type="reg"/> documentation.  
//      See these sources for detailed information regarding the  
//      Microsoft samples programs.  
  
//  
  
#undef UNICODE  
  
#define WIN32_LEAN_AND_MEAN  
  
#include <windows.h>  
#include <winsock2.h>  
#include <ws2tcpip.h>  
#include <mstcpip.h>  
#include <stdlib.h>  
#include <stdio.h>  
  
// Needed for the Windows 2000 IPv6 Tech Preview.  
#if (_WIN32_WINNT == 0x0500)  
#include <tpipv6.h>  
#endif  
  
// Link with ws2_32.lib  
#pragma comment(lib, "Ws2_32.lib")  
  
#define STRICMP _stricmp  
  
//  
// This code assumes that at the transport level, the system only supports  
// one stream protocol (TCP) and one datagram protocol (UDP). Therefore,  
// specifying a socket type of SOCK_STREAM is equivalent to specifying TCP  
// and specifying a socket type of SOCK_DGRAM is equivalent to specifying UDP.  
//  
  
#define DEFAULT_FAMILY      PF_UNSPEC    // Accept either IPv4 or IPv6  
#define DEFAULT_SOCKTYPE    SOCK_STREAM  // TCP  
#define DEFAULT_PORT         "5001"       // Arbitrary, albeit a historical test port  
#define BUFFER_SIZE          64          // Set very small for demonstration purposes  
  
void Usage(char *ProgName)  
{  
    fprintf(stderr, "\nSimple socket sample server program.\n");  
    fprintf(stderr,  
        "\n%[ -f family] [-t transport] [-p port] [-a address]\n\n",  
        ProgName);  
    fprintf(stderr,  
        "      family\tOne of PF_INET, PF_INET6 or PF_UNSPEC. (default %s)\n",  
        (DEFAULT_FAMILY ==  
            PF_UNSPEC) ? "PF_UNSPEC" : ((DEFAULT_FAMILY ==  
                PF_INET) ? "PF_INET" : "PF_INET6"));  
    fprintf(stderr, "      transport\tEither TCP or UDP. (default: %s)\n",
```

```

        (DEFAULT_SOCKTYPE == SOCK_STREAM) ? "TCP" : "UDP");
    fprintf(stderr, " port\t\tPort on which to bind. (default %s)\n",
            DEFAULT_PORT);
    fprintf(stderr,
            " address\tIP address on which to bind. (default: unspecified address)\n");
    WSACleanup();
    exit(1);
}

LPSTR PrintError(int ErrorCode)
{
    static char Message[1024];

    // If this program was multithreaded, we'd want to use
    // FORMAT_MESSAGE_ALLOCATE_BUFFER instead of a static buffer here.
    // (And of course, free the buffer when we were done with it)

    FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS |
                  FORMAT_MESSAGE_MAX_WIDTH_MASK, NULL, ErrorCode,
                  MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                  (LPSTR) Message, 1024, NULL);

    return Message;
}

int main(int argc, char **argv)
{
    char Buffer[BUFFER_SIZE], Hostname[NI_MAXHOST];
    int Family = DEFAULT_FAMILY;
    int SocketType = DEFAULT_SOCKTYPE;
    char *Port = DEFAULT_PORT;
    char *Address = NULL;
    int i, NumSocks, RetVal, FromLen, AmountRead;
//    int idx;
    SOCKADDR_STORAGE From;
    WSADATA wsaData;
    ADDRINFO Hints, *AddrInfo, *AI;
    SOCKET ServSock[FD_SETSIZE];
    fd_set SockSet;

    // Parse arguments
    if (argc > 1) {
        for (i = 1; i < argc; i++) {
            if ((argv[i][0] == '-') || (argv[i][0] == '/') &&
                (argv[i][1] != 0) && (argv[i][2] == 0)) {
                switch (tolower(argv[i][1])) {
                    case 'f':
                        if (!argv[i + 1])
                            Usage(argv[0]);
                        if (!STRICMP(argv[i + 1], "PF_INET"))
                            Family = PF_INET;
                        else if (!STRICMP(argv[i + 1], "PF_INET6"))
                            Family = PF_INET6;
                        else if (!STRICMP(argv[i + 1], "PF_UNSPEC"))
                            Family = PF_UNSPEC;
                        else
                            Usage(argv[0]);
                        i++;
                        break;

                    case 't':
                        if (!argv[i + 1])
                            Usage(argv[0]);
                        if (!STRICMP(argv[i + 1], "TCP"))
                            SocketType = SOCK_STREAM;
                        else if (!STRICMP(argv[i + 1], "UDP"))
                            SocketType = SOCK_DGRAM;
                        else
                            Usage(argv[0]);
                        i++;
                }
            }
        }
    }
}

```

```

        break;

    case 'a':
        if (argv[i + 1]) {
            if (argv[i + 1][0] != '-') {
                Address = argv[++i];
                break;
            }
        }
        Usage(argv[0]);
        break;

    case 'p':
        if (argv[i + 1]) {
            if (argv[i + 1][0] != '-') {
                Port = argv[++i];
                break;
            }
        }
        Usage(argv[0]);
        break;

    default:
        Usage(argv[0]);
        break;
    }
} else
    Usage(argv[0]);
}

// Ask for Winsock version 2.2.
if ((RetVal = WSAStartup(MAKEWORD(2, 2), &wsaData)) != 0) {
    fprintf(stderr, "WSAStartup failed with error %d: %s\n",
            RetVal, PrintError(RetVal));
    WSACleanup();
    return -1;
}

if (Port == NULL) {
    Usage(argv[0]);
}
//
// By setting the AI_PASSIVE flag in the hints to getaddrinfo, we're
// indicating that we intend to use the resulting address(es) to bind
// to a socket(s) for accepting incoming connections. This means that
// when the Address parameter is NULL, getaddrinfo will return one
// entry per allowed protocol family containing the unspecified address
// for that family.
//
memset(&Hints, 0, sizeof (Hints));
Hints.ai_family = Family;
Hints.ai_socktype = SocketType;
Hints.ai_flags = AI_NUMERICHOST | AI_PASSIVE;
RetVal = getaddrinfo(Address, Port, &Hints, &AddrInfo);
if (RetVal != 0) {
    fprintf(stderr, "getaddrinfo failed with error %d: %s\n",
            RetVal, gai_strerror(RetVal));
    WSACleanup();
    return -1;
}
//
// For each address getaddrinfo returned, we create a new socket,
// bind that address to it, and create a queue to listen on.
//
for (i = 0, AI = AddrInfo; AI != NULL; AI = AI->ai_next) {

    // Highly unlikely, but check anyway.
    if (i == FD_SETSIZE) {
        printf("getaddrinfo returned more addresses than we could use.\n");
}

```

```

        break;
    }
    // This example only supports PF_INET and PF_INET6.
    if ((AI->ai_family != PF_INET) && (AI->ai_family != PF_INET6))
        continue;

    // Open a socket with the correct address family for this address.
    ServSock[i] = socket(AI->ai_family, AI->ai_socktype, AI->ai_protocol);
    if (ServSock[i] == INVALID_SOCKET) {
        fprintf(stderr, "socket() failed with error %d: %s\n",
                WSAGetLastError(), PrintError(WSAGetLastError()));
        continue;
    }

    if ((AI->ai_family == PF_INET6) &&
        IN6_IS_ADDR_LINKLOCAL((IN6_ADDR *) INETADDR_ADDRESS(AI->ai_addr)) &&
        (((SOCKADDR_IN6 *) (AI->ai_addr))->sin6_scope_id == 0)
    ) {
        fprintf(stderr,
                "IPv6 link local addresses should specify a scope ID!\n");
    }
    //
    // bind() associates a local address and port combination
    // with the socket just created. This is most useful when
    // the application is a server that has a well-known port
    // that clients know about in advance.
    //
    if (bind(ServSock[i], AI->ai_addr, (int) AI->ai_addrlen) == SOCKET_ERROR) {
        fprintf(stderr, "bind() failed with error %d: %s\n",
                WSAGetLastError(), PrintError(WSAGetLastError()));
        closesocket(ServSock[i]);
        continue;
    }
    //
    // So far, everything we did was applicable to TCP as well as UDP.
    // However, there are certain fundamental differences between stream
    // protocols such as TCP and datagram protocols such as UDP.
    //
    // Only connection orientated sockets, for example those of type
    // SOCK_STREAM, can listen() for incoming connections.
    //
    if (SocketType == SOCK_STREAM) {
        if (listen(ServSock[i], 5) == SOCKET_ERROR) {
            fprintf(stderr, "listen() failed with error %d: %s\n",
                    WSAGetLastError(), PrintError(WSAGetLastError()));
            closesocket(ServSock[i]);
            continue;
        }
    }
}

printf("'Listening' on port %s, protocol %s, protocol family %s\n",
       Port, (SocketType == SOCK_STREAM) ? "TCP" : "UDP",
       (AI->ai_family == PF_INET) ? "PF_INET" : "PF_INET6");
i++;
}

freeaddrinfo(AddrInfo);

if (i == 0) {
    fprintf(stderr, "Fatal error: unable to serve on any address.\n");
    WSACleanup();
    return -1;
}
NumSocks = i;

//
// We now put the server into an eternal loop,
// serving requests as they arrive.
//

```

```

    FD_ZERO(&SockSet);
    while (1) {

        FromLen = sizeof (From);

        //
        // For connection orientated protocols, we will handle the
        // packets received from a connection collectively. For datagram
        // protocols, we have to handle each datagram individually.
        //

        //
        // Check to see if we have any sockets remaining to be served
        // from previous time through this loop. If not, call select()
        // to wait for a connection request or a datagram to arrive.
        //

        for (i = 0; i < NumSocks; i++) {
            if (FD_ISSET(ServSock[i], &SockSet))
                break;
        }

        if (i == NumSocks) {
            for (i = 0; i < NumSocks; i++)
                FD_SET(ServSock[i], &SockSet);
            if (select(NumSocks, &SockSet, 0, 0, 0) == SOCKET_ERROR) {
                fprintf(stderr, "select() failed with error %d: %s\n",
                        WSAGetLastError(), PrintError(WSAGetLastError()));
                WSACleanup();
                return -1;
            }
        }

        for (i = 0; i < NumSocks; i++) {
            if (FD_ISSET(ServSock[i], &SockSet)) {
                FD_CLR(ServSock[i], &SockSet);
                break;
            }
        }
    }

    if (SocketType == SOCK_STREAM) {
        SOCKET ConnSock;

        //
        // Since this socket was returned by the select(), we know we
        // have a connection waiting and that this accept() won't block.
        //

        ConnSock = accept(ServSock[i], (LPSOCKADDR) & From, &FromLen);
        if (ConnSock == INVALID_SOCKET) {
            fprintf(stderr, "accept() failed with error %d: %s\n",
                    WSAGetLastError(), PrintError(WSAGetLastError()));
            WSACleanup();
            return -1;
        }

        if (getnameinfo((LPSOCKADDR) & From, FromLen, Hostname,
                        sizeof (Hostname), NULL, 0, NI_NUMERICHOST) != 0)
            strcpy_s(Hostname, NI_MAXHOST, "<unknown>");
        printf("\nAccepted connection from %s\n", Hostname);

        //
        // This sample server only handles connections sequentially.
        // To handle multiple connections simultaneously, a server
        // would likely want to launch another thread or process at this
        // point to handle each individual connection. Alternatively,
        // it could keep a socket per connection and use select()
        // on the fd_set to determine which to read from next.
        //

        // Here we just loop until this connection terminates.
        //

        while (1) {

```

```

        //
        // We now read in data from the client. Because TCP
        // does NOT maintain message boundaries, we may recv()
        // the client's data grouped differently than it was
        // sent. Since all this server does is echo the data it
        // receives back to the client, we don't need to concern
        // ourselves about message boundaries. But it does mean
        // that the message data we print for a particular recv()
        // below may contain more or less data than was contained
        // in a particular client send().
        //

        AmountRead = recv(ConnSock, Buffer, sizeof (Buffer), 0);
        if (AmountRead == SOCKET_ERROR) {
            fprintf(stderr, "recv() failed with error %d: %s\n",
                    WSAGetLastError(), PrintError(WSAGetLastError()));
            closesocket(ConnSock);
            break;
        }
        if (AmountRead == 0) {
            printf("Client closed connection\n");
            closesocket(ConnSock);
            break;
        }

        printf("Received %d bytes from client: [%.*s]\n",
               AmountRead, AmountRead, Buffer);
        printf("Echoing same data back to client\n");

        RetVal = send(ConnSock, Buffer, AmountRead, 0);
        if (RetVal == SOCKET_ERROR) {
            fprintf(stderr, "send() failed: error %d: %s\n",
                    WSAGetLastError(), PrintError(WSAGetLastError()));
            closesocket(ConnSock);
            break;
        }
    }

} else {

    //
    // Since UDP maintains message boundaries, the amount of data
    // we get from a recvfrom() should match exactly the amount of
    // data the client sent in the corresponding sendto().
    //

    AmountRead = recvfrom(ServSock[i], Buffer, sizeof (Buffer), 0,
                          (LPSOCKADDR) & From, &FromLen);
    if (AmountRead == SOCKET_ERROR) {
        fprintf(stderr, "recvfrom() failed with error %d: %s\n",
                WSAGetLastError(), PrintError(WSAGetLastError()));
        closesocket(ServSock[i]);
        break;
    }
    if (AmountRead == 0) {
        // This should never happen on an unconnected socket, but...
        printf("recvfrom() returned zero, aborting\n");
        closesocket(ServSock[i]);
        break;
    }

    RetVal = getnameinfo((LPSOCKADDR) & From, FromLen, Hostname,
                         sizeof (Hostname), NULL, 0, NI_NUMERICHOST);
    if (RetVal != 0) {
        fprintf(stderr, "getnameinfo() failed with error %d: %s\n",
                RetVal, PrintError(RetVal));
        strcpy_s(Hostname, NI_MAXHOST, "<unknown>");
    }
}

```

```
    printf("Received a %d byte datagram from %s: [%.*s]\n",
           AmountRead, Hostname, AmountRead, Buffer);
    printf("Echoing same data back to client\n");

    RetVal = sendto(ServSock[i], Buffer, AmountRead, 0,
                    (LPSOCKADDR) & From, FromLen);
    if (RetVal == SOCKET_ERROR) {
        fprintf(stderr, "send() failed with error %d: %s\n",
                WSAGetLastError(), PrintError(WSAGetLastError()));
    }
}

return 0;
}
```

# High-performance Windows Sockets Applications

3/5/2021 • 2 minutes to read • [Edit Online](#)

Microsoft Windows networking components have been developed for performance and scalability. This enables applications to maximize available network bandwidth. Windows Sockets and the Windows TCP/IP protocol stack have been tuned and streamlined. As a result, properly written Windows applications can achieve exceptional throughput and performance, as the following facts illustrate:

- Windows is capable of servicing over 200,000 simultaneous TCP connections.
- In a test conducted by SPECWeb96, Internet Information Server on Windows serviced over 25,000 HTTP requests per second.
- Windows set a transmission record of over 750Mbps on a transcontinental gigabit network consisting of 10 hops.

These achievements illustrate that Windows TCP/IP processes data very quickly. Many applications, however, do not take advantage of the Windows, TCP/IP, and Windows Sockets performance capabilities because they unknowingly implement performance-hampering techniques.

In this guide, you will learn to identify common programming mistakes and how to avoid them. Then, you will learn techniques that enable Windows Sockets applications to perform optimally. This guide is presented in six sections. The order of the sections is intentional; to get the most out of this guide, read it in order. The following table provides links to each section, as well as a brief description of each topic.

TOPIC	DESCRIPTION
<a href="#">Network Terminology</a>	Defines networking terminology and metrics necessary to understanding the performance of a network application.
<a href="#">Performance Dimensions</a>	Discusses performance dimensions that affect the perceived and actual network performance of an application.
<a href="#">TCP/IP Characteristics</a>	Defines TCP/IP protocol characteristics that can result in performance issues for a poorly written application.
<a href="#">Application Behavior</a>	Explains how to recognize the signs of a poorly performing network application.
<a href="#">Improving a Slow Application</a>	Provides samples of application design issues that contribute to a poorly performing application, and makes changes to code to improve performance.
<a href="#">Best Practices for Interactive Applications</a>	Lists the best practices to employ for developing optimal interactive network applications.

# Network Terminology (Windows Sockets 2)

3/5/2021 • 2 minutes to read • [Edit Online](#)

Metrics are used to measure aspects of network and protocol performance. The values for such metrics in various scenarios indicate the level of performance of a network application. This section defines terms and metrics used industry-wide for measuring network application performance. These terms are used throughout the rest of this guide.

- Round Trip Time (RTT)

Time, in milliseconds, for a request to make a trip from a source computer to a destination computer, and back again. Lower values indicate better performance. Forward and return path times are not necessarily equal.

RTT values are affected by network infrastructure, distance between nodes, network conditions, and packet size. Packet size, congestion and payload compressibility impact RTT when measured on slow links, such as dial-up connections. Other factors affect RTT, including forward error correction and data compression, which introduce buffers and queues that increase RTT, and therefore decrease performance.

- Goodput

Measure of useful application data successfully processed by the receiver, in bits-per-second. Goodput enables the measurement of effective or useful throughput and includes only application data; packet, protocol, and media headers are considered overhead, and are not part of goodput.

- Protocol Overhead

Nonapplication bytes, including protocol and media framing, divided by the total number of bytes transmitted. The value is expressed as a percentage. Higher values indicate poorer performance.

Overhead is calculated for both directions in this guide, but can be calculated for each direction separately.

- Bandwidth-Delay Product

Product of the bits-per-second bandwidth of the network, and the RTT (in seconds). This value equates to the number of bits it takes to fill available network bandwidth. When the value for bandwidth-delay product is high, the TCP/IP stack must handle large amounts of unacknowledged data in order to keep the pipeline full. Bandwidth-delay product is a key end-to-end metric for streaming applications.

# Performance Dimensions

3/5/2021 • 2 minutes to read • [Edit Online](#)

Application performance differs according to perspective. For a user, an ideal application would deliver high performance and responsiveness. For an administrator, an ideal application would consume the least amount of network resources. Performance also differs based on application type. Performance for a bulk file transfer application might have different implications than performance for a transactional application.

The following sections address differing performance dimensions and application types, and how networking environments affects performance.

- [Performance Needs: Users and Administrators](#)
- [Transactional versus Streaming Applications](#)
- [Different Network Environments](#)

## Related topics

[High-performance Windows Sockets Applications](#)

# Performance Needs: Users and Administrators

3/5/2021 • 2 minutes to read • [Edit Online](#)

Users judge application performance by their experience:

- Is the application quick to respond?
- Is an hourglass icon displayed while background operations are carried out?
- Does the application launch and close quickly?
- Are errors handled in an understandable way?

To summarize, users want applications to be fast and predictable.

In contrast, administrators often judge an application's performance on how efficiently it uses network resources. Administrators may ask:

- Does the application have low overhead and efficient network usage?
- Are the minimum number of connections used, so my server can service as many users at once as possible?
- Am I constantly calling helpdesk?

In short, administrators want applications to scale.

## Best Practices for Performance Needs

When developing a Windows Sockets application, these performance requirements translate into useful rules.

- Have network applications initialize quickly.

The user interface should not have to wait for network responses. Some tasks can be performed before the network is available, or without the network. If the network is not responding, the user may need the user interface for simple operations, such as closing the application.

- Do not wait for the network for shutdown.

Properly written client-server applications handle abortive disconnects gracefully. Do not initiate a potentially lengthy operation, such as synchronizing files or folders with a server, that cannot be interrupted on shutdown. Networks are not consistently responsive, so even small operations could prove time consuming. Provide positive feedback for users, including indications of progress and estimated completion times.

- Ensure a responsive user interface.

Application responsiveness helps eliminate unnecessary helpdesk calls. A good guideline for interactive response is 500 milliseconds. Users perceive pauses longer than 500 milliseconds as a lag in performance. Applications should be responsive enough to provide the user with confidence about the application.

- Scrutinize network errors.

Not all network errors are critical. For example, an application that has received or posted all of its data can likely ignore errors when closing the connection. Do not assume the network or the user is available; either handle errors without user intervention, or ignore them if errors are noncritical.

- An application should define its own reasonable time outs.

For example, a Windows Sockets connect() request may block under some conditions for as much as 21 seconds. Applications may need to introduce their own time outs as appropriate for their users.

- Minimize protocol overhead.

Conserving network bandwidth is partially about minimizing the protocol overhead incurred by your application. It is also about eliminating unnecessary network traffic. Protocols with a lower header overhead can be used to transfer application data. For example, when sending smaller amounts of noncritical or repeatable data, use UDP as opposed to TCP to reduce the overhead associated with connection establishment and maintenance. If the same data must be sent to multiple recipients, consider multicast. Be aware that UDP applications are not flow-controlled—pushing beyond the available bandwidth can cause catastrophic network failure. The Netstat utility can be used with its -e and -s options to display statistics for various protocols.

- Conserve system resources.

System resources can be consumed quickly if proper restraint is not used. For example, sockets and TCP connections consume resources. Do not use several TCP connections per client where one will serve the application's purpose.

For transactional applications, good user experience and low network utilization are not conflicting goals. The network is a bottleneck. Network-intensive applications spend more time waiting, and well written network applications are designed to minimize unnecessary wait time, both for the user interface and for network transmissions.

## Related topics

[High-performance Windows Sockets Applications](#)

[Performance Dimensions](#)

# Transactional versus Streaming Applications

3/5/2021 • 2 minutes to read • [Edit Online](#)

There are two fundamental types of network applications: *transactional* and *streaming*. These application types are also called *interactive* and *batch processing* application types, respectively.

Transactional applications are stop-and-go applications. They usually perform request/reply operations, often ordered. Examples of transactional applications include synchronous remote procedure call (RPC), as well as some HTTP and Domain Name System (DNS) implementations.

Streaming applications move data. To describe streaming applications with a parallel term, streaming applications adhere to a pedal-to-the-metal data transmission philosophy, usually with little concern for data ordering. Examples of streaming applications include network backup and file transfer protocol (FTP).

Once the application type is determined, its network and protocol characteristics are determined as well.

## Related topics

[High-performance Windows Sockets Applications](#)

[Performance Dimensions](#)

# Different Network Environments

3/5/2021 • 2 minutes to read • [Edit Online](#)

Several network environments affect the networked behavior of an application. Properties that differentiate network environments include low versus high bandwidth, and low versus high RTT. Network environments affect transactional and streaming applications in different ways. Transactional applications are more sensitive to RTT; streaming applications are more sensitive to bandwidth-delay products.

	Low Bandwidth	High Bandwidth
Low RTT		LAN
High RTT	Dial-up WAN	Wireless Satellite WAN

Dial-up networks and some wireless networks have a variable RTT. Satellite networks generally have an asymmetric bandwidth between upstream and downstream. Wireless LAN and xDSL are good examples of networks with bandwidth-delay products similar to that of Fast Ethernet.

## Related topics

[High-performance Windows Sockets Applications](#)

[Performance Dimensions](#)

# TCP/IP Characteristics

3/5/2021 • 2 minutes to read • [Edit Online](#)

TCP/IP has characteristics that enable the protocol to operate as its standardized implementation requirements dictate. These characteristics can combine with development choices that result in poor performance. The impact these TCP/IP characteristics have on an application depend on whether the application is transactional or streaming.

Transactional applications are affected by the overhead required for connection establishment and termination. For example, each time a connection is established on an Ethernet network, three packets of approximately 60 bytes each must be sent, and approximately one RTT is required for the exchange. When termination of a connection occurs, four packets are exchanged. This is for each connection; an application that opens and closes connections often incurs this overhead on each occurrence.

Another aspect of TCP/IP is *slow-start*, which takes place whenever a connection is established. Slow-start is an artificial limit on the number of data segments that can be sent before acknowledgment of those segments is received. Slow-start is designed to limit network congestion. When a connection over Ethernet is established, regardless of the receiver's window size, a 4 KB transmission can take up to 3-4 RTT due to slow-start.

A TCP/IP optimization called the Nagle Algorithm can also limit data transfer speed on a connection. The Nagle Algorithm is designed to reduce protocol overhead for applications that send small amounts of data, such as Telnet, which sends a single character at a time. Rather than immediately send a packet with lots of header and little data, the stack waits for more data from the application, or an acknowledgment, before proceeding.

Delayed acknowledgments, commonly referred to as *delayed ACK*, are also designed into TCP/IP to enable more efficient piggybacking of acknowledgments when return data is forthcoming from the receiving side application. Unfortunately, if this data is not forthcoming and the sending side is waiting for an acknowledgment, delays of approximately 200 milliseconds per send can occur.

When a TCP connection is closed, connection resources at the node that initiated the close are put into a wait state, called TIME-WAIT, to guard against data corruption if duplicate packets linger in the network. This ensures both ends are finished with the connection. This can cause depletion of resources required per-connection, such as RAM and ports, when applications open and close connections frequently.

Besides being affected by delayed ACK and other congestion avoidance schemes, streaming applications can also be affected by a too-small default receive window size on the receiving end.

## Related topics

[Application Behavior](#)

[High-performance Windows Sockets Applications](#)

[Nagle Algorithm](#)

# Application Behavior

3/5/2021 • 2 minutes to read • [Edit Online](#)

Another aspect of application development to consider is the difference in behavior between local, or intracomputer operations, and behavior when operations take place between two networked computers. There are application behaviors that may work acceptably well on a local computer, but when run across a network, cause significant performance degradation and resource consumption. The following application behaviors should be avoided when developing Windows Sockets applications.

## Behaviors to Avoid

- Chatty Applications.

Some applications perform many small transactions. When combined with the network overhead associated with each such transaction, the effect is multiplied. In networking, small transactions can consume as many resources and as much time as large transactions. A better approach is to combine many small transactions into a single large transaction.

- Serialized Transactions.

Unnecessary serialization of transactions can result in poor performance, and affect scalability. For example, 1000 serialized transactions take at least  $1000 * \text{RTT}$  to complete. A better approach is to run unrelated transactions in parallel. When serialized applications are combined with chatty applications, responsiveness can be seriously hampered.

### NOTE

Properly deserializing an application can be difficult. If changing from serialized to parallel proves too difficult, consider combining multiple transactions into fewer large transactions.

- Fat Transactions.

Applications that send unnecessary bytes on the network are considered fat applications. Applications that send unnecessary bytes on the network increase network overhead, and performance is hampered. This situation often comes from inefficient data structures or inefficient data streaming. To remedy this, optimize data structures, or consider using compression.

The following sections address aspects of application development to consider.

- [TCP/IP-specific Issues](#)
- [Recognizing Slow Applications](#)
- [Calculating Overhead with Netstat](#)

## Related topics

[High-performance Windows Sockets Applications](#)

# TCP/IP-specific Issues

3/5/2021 • 2 minutes to read • [Edit Online](#)

Certain programming techniques run into performance issues that are linked to the implementation of TCP/IP. Such performance issues do not suggest that TCP/IP is inefficient or a performance bottleneck; rather, these issues are seen when TCP/IP operations are not understood.

The following issues identify common scenarios in which the combination of TCP/IP operation and network application development choices result in poor performance.

- Connect-heavy Applications.

Some applications instantiate a new TCP connection for each transaction. TCP connection establishment takes time, contributes extra RTTs, and is subject to slow-start. In addition, the closed connections are subject to TIME-WAIT, which consumes system resources.

Connect-heavy applications are common largely because they are easy to create; testing and error handling is very simple. Detecting faults on a persistent connection can take considerable code and effort, and therefore is sometimes not completed.

Remedy this situation by reusing the TCP connection. This may cause serialization over the TCP connection unless the transactions are multiplexed over multiple connections. If this approach is taken, the number of connections should be limited to two, and application-layer framing and advanced error handling are required.

- Zero-length Send Buffers and Blocking Sends.

Turning off buffering by using the **setsockopt** function to set the send buffer (SO\_SNDBUF) to zero is similar to turning off disk caching. When setting the send buffer to zero and issuing blocking sends, an application has a fifty percent chance of hitting a 200-millisecond delayed acknowledgment.

Do not turn off send buffering unless you have considered the impact in all network environments. One exception: streaming data using overlapped I/O should set the send buffer to zero.

- Send-Send-Receive programming model.

Structuring an application to perform send-send-receives increases your chances of encountering the Nagle Algorithm, which causes a delay of RTT+200 ms. The Nagle Algorithm may be encountered if the last send is less than the TCP Maximum Segment Size (MSS, the maximum data in a single datagram). MSS can be a very large value (64K in IPv4, and even larger in IPv6), so do not count on a typically small MSS. A better option is to combine the two sends into a single send using the **WSASend** or **memcpy** function.

- Large Number of Simultaneous Connections.

Concurrent connections should not exceed two, except in special purpose applications. Exceeding two concurrent connections results in wasted resources. A good rule is to have up to four short lived connections, or two persistent connections per destination.

## Related topics

[Application Behavior](#)

[High-performance Windows Sockets Applications](#)

## Nagle Algorithm

# Recognizing Slow Applications

3/5/2021 • 2 minutes to read • [Edit Online](#)

This guide identifies a *slow* application as a Microsoft Windows application with impaired performance. A slow application exhibits one or more of the following symptoms:

- CPU and network utilization are low.

The computer appears to be waiting on something. Often, the application is waiting on the network.

- Turning off the Nagle Algorithm through the TCP\_NODELAY socket option increases performance.

This indicates other problems, and should not be considered a solution. Turning off the Nagle algorithm increases the protocol overhead. Do not use this method as a fix for the broken applications—only as an indication the application needs other work to fix performance issues.

- The application exhibits high overhead.

To calculate your applications overhead, determine how much data you intended to transfer in each direction. Then use Netstat and add (for Ethernet) 60 bytes for each packet and 500 bytes for each connection. The best overhead that can be expected for streaming over Ethernet is approximately 6%. For a modem connection, the best overhead is approximately 2%, due to the fact that a PPP link uses header compression. See [Calculating Overhead with Netstat](#) for more information.

- Application response slows when the connection has a large RTT.

Assuming the application is not approaching the link's bandwidth, a large RTT should have little or no effect. A dramatic slowdown with a large RTT is a clear sign of serialized processing and many small transactions.

Every application should be tested in an environment with a large RTT. Doing so reveals most applications that suffer from poor development choices. This testing can be performed in several environments, including a wireless LAN network, a link-delay simulator, or a satellite network.

## Related topics

[Application Behavior](#)

[High-performance Windows Sockets Applications](#)

[Nagle Algorithm](#)

# Calculating Overhead with Netstat

3/5/2021 • 2 minutes to read • [Edit Online](#)

Calculating overhead with Netstat should be performed on a quiet network to avoid other network traffic from skewing the data, such as broadcast or multicast traffic.

## To calculate an application's network overhead using Netstat

1. Retrieve the current interface statistics using Netstat.
2. Execute the application.
3. Get the interface statistics, again using Netstat.
4. Calculate the number of bytes received between the two Netstat measurements.

## Example

The following example demonstrates these steps, using TTCP to send 10 bytes of data, one byte at a time.

First, a theoretical overhead for the application is calculated. For this test, all packets are 60 bytes (the Ethernet minimum). This transfer requires three packets to set up the connection, 10 data packets, 10 acknowledgment packets (delayed ACK is triggered nearly every time), and four packets to close the connection gracefully.

This equates to 27 packets of 60 bytes each, or  $1620$  bytes ( $3+4+10+10)*60=1620$ ). Since only 10 bytes of data are transferred, the overhead is 1610 bytes, which is over 99% protocol overhead.

## Commands

**netstat -e**

Interface Statistics		
	Received	Sent
Bytes	392291400	444684566
Unicast packets	487627	570086
Non-unicast packets	1159163	11300
Discards	0	0
Errors	0	0
Unknown protocols	52812	

**ttcp -t -h0 -D -l1 -n10 -p9 172.31.71.99**

```
ttcp-t: 10 bytes in 2168 real milliseconds = 0 KB/sec
ttcp-t: 10 I/O calls, msec/call = 216, calls/sec = 4, bytes/call = 1
```

**netstat -e**

Interface Statistics		
	Received	Sent
Bytes	39229207	444685382
Unicast packets	487641	570100
Non-unicast packets	1159164	11301
Discards	0	0
Errors	0	0
Unknown protocols	52812	

## Calculations

**Sent:** 816 bytes

**Received:** 674 bytes

**Total bytes:** 1490

**User bytes:** 10

**Overhead:**  $1480/1490 = 99.3\%$

**Goodput:** \*\*= 5 bytes/second (or 40 bits/s)

**NOTE**

Actual bytes transferred do not match the theoretical values due to padding bytes not being accounted for in the Netstat values.

## Related topics

[Application Behavior](#)

[High-performance Windows Sockets Applications](#)

# Improving a Slow Application

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section examines a portion of a sample application that operates over the network very slowly. Throughout this section, modifications are made to the initial code to improve its performance.

The mock sample is the updated portion for a game called Life. The application is written such that the client performs the calculations for the updates and sends them to the server. The server then displays the resulting Life field. The output from the client is a stream of bytes, grouped in threes (triplets), each triplet representing one cell update. The bytes in the triplet represent the row, column, and value, respectively, for the cell.

This sample begins as an intentionally poor performing application, which provides the baseline from which performance improvements can be illustrated. From there, the code is improved three times to address various issues that affect performance. These samples should be read in order, as each iteration improves on the previous version.

The baseline code, and the revisions that improve that code, are the following:

- [The Baseline Version: A Very Poor Performing Application](#)
- [Revision 1: Cleaning up the Obvious](#)
- [Revision 2: Redesigning for Fewer Connects](#)
- [Revision 3: Compressed Block Send](#)
- [Future Improvements](#)

## WARNING

The first few examples of the application provide intentionally poor performance, in order to illustrate performance improvements possible with changes to code. Do not use these code samples in your application; they are for illustration purposes only.

## Related topics

[High-performance Windows Sockets Applications](#)

# The Baseline Version: A Very Poor Performing Application

3/5/2021 • 2 minutes to read • [Edit Online](#)

The initial, poor performing code sample used to calculate the updates is as follows:

## NOTE

For simplicity, there is no error handling in the following examples. Any production application always checks return values.

## WARNING

The first few examples of the application provide intentionally poor performance, in order to illustrate performance improvements possible with changes to code. Do not use these code samples in your application; they are for illustration purposes only.

```
#include <windows.h>

BOOL Map[ROWS][COLS];

void LifeUpdate()
{
    ComputeNext( Map );
    for( int i = 0 ; i < ROWS ; ++i )      //serialized
        for( int j = 0 ; j < COLS ; ++j )
            Set( i, j, Map[i][j] );      //chatty
}

BYTE Set(row, col, bAlive)
{
    SOCKET s = socket(...);
    BYTE byRet = 0;
    setsockopt( s, SO_SNDBUF, &Zero, sizeof(int) );
    bind( s, ... );
    connect( s, ... );
    send( s, &row, 1 );
    send( s, &col, 1 );
    send( s, &bAlive, 1 );
    recv( s, &byRet, 1 );
    closesocket( s );
    return byRet;
}
```

In this state, the application has the worst possible network performance. The problems with this version of the sample application include:

- The application is chatty. Each transaction is too small — cells do not need to be updated one by one.
- The transactions are strictly serialized, even though the cells could be updated concurrently.

- The send buffer is set to zero, and the application incurs a 200-millisecond delay for each send — three times per cell.
- The application is very connect heavy, connecting once for each cell. Applications are limited in the number of connections per second for a given destination because of TIME-WAIT state, but that is not an issue here, since each transaction takes over 600 milliseconds.
- The application is fat; many transactions have no effect on the server state, because many cells do not change from update to update.
- The application exhibits poor streaming; small sends consume a lot of CPU and RAM.
- The application assumes little-endian representation for its sends. This is a natural assumption for the current Windows platform, but can be dangerous for long-lived code.

## Key Performance Metrics

The following performance metrics are expressed in Round Trip Time (RTT), Goodput, and Protocol Overhead. See the [Network Terminology](#) topic for an explanation of these terms.

- Cell time, network time for a single cell update, requires  $4 \times \text{RTT} + 600$  milliseconds for Nagle and delayed ACK interactions.
- Goodput is less than 6 bytes.
- Protocol Overhead is 99.6%.

## Related topics

[Improving a Slow Application](#)

[Network Terminology](#)

[Revision 1: Cleaning up the Obvious](#)

[Revision 2: Redesigning for Fewer Connects](#)

[Revision 3: Compressed Block Send](#)

[Future Improvements](#)

# Revision One: Cleaning up the Obvious

3/5/2021 • 2 minutes to read • [Edit Online](#)

In this version of the example program, a few obvious changes have been made that will take initial strides at improving performance. This version of the code is far from performance-tuned, but it is a good incremental step that enables examination of the effects of incrementally better approaches.

## WARNING

This example of the application provides intentionally poor performance, in order to illustrate performance improvements possible with changes to code. Do not use this code sample in your application; it is for illustration purposes only.

```
#include <windows.h>

BYTE Set(row, col, bAlive)
{
    SOCKET s = socket(...);
    BYTE byRet = 0;
    BYTE tmp[3];
    tmp[0] = (BYTE)row;
    tmp[1] = (BYTE)col;
    tmp[2] = (BYTE)bAlive;
    bind( s, ... );
    connect( s, ... );
    send( s, &tmp, 3 );
    recv( s, &byRet, 1 );
    closesocket( s );
    return byRet;
}
```

## Changes in this Version

This version reflects the following changes:

- Removed SNDBUF=0. The 200 millisecond delays due to unbuffered sends and delayed acknowledgments are removed.
- Removed the Send-Send-Receive behavior. This change eliminates 200-millisecond delays due to Nagle and delayed ACK interactions.
- Removed little-endian assumption. The bytes are now in order.

## Remaining Problems

In this version, the following problems remain:

- The application is still connect heavy. Since the 600+ millisecond delays are removed, the application now hits the 12 connects-per-second sustained rate. Many concurrent connections now becomes an issue.
- The application is still fat; unchanged cells are still propagated to the server.
- The application still exhibits poor streaming; 3-byte sends are still small sends.
- The application now sends 2 bytes/RTT, which equals 4 KB/s on directly connected Ethernet. This is not too fast, but TIME-WAIT will likely cause problems first.

- The overhead is down to 99.3%, which is still not a good overhead percentage.

## Key Performance Metrics

The following key performance metrics are expressed in Round Trip Time (RTT), Goodput, and Protocol Overhead. See the [Network Terminology](#) topic for an explanation of these terms.

This version reflect the following performance metrics:

- Cell Time— $2 \times \text{RTT}$
- Goodput—2 bytes/RTT
- Protocol Overhead—99.3 percent

## Related topics

[Improving a Slow Application](#)

[Network Terminology](#)

[The Baseline Version: A Very Poor Performing Application](#)

[Revision 2: Redesigning for Fewer Connects](#)

[Revision 3: Compressed Block Send](#)

[Future Improvements](#)

# Revision Two: Redesigning for Fewer Connects

3/5/2021 • 2 minutes to read • [Edit Online](#)

In this revision, the sample application is redesigned to eliminate unnecessary connects.

## WARNING

This examples of the application also provide intentionally poor performance, in order to illustrate performance improvements possible with changes to code. Do not use this code sample in your application; it is for illustration purposes only.

```
ComputeNext( Map );
bind( s, ... );
connect( s, ... );
for(int i=0 ; i < ROWS ; ++i)
    for(int j=0 ; j < COLS ; ++j)
    {
        BYTE tmp[3];
        tmp[0] = i;
        tmp[1] = j;
        tmp[2] = Map[i][j];
        send( s, tmp, 3 );
        recv( s, &byRet, 1 );
    }
closesocket( s );
```

## Remaining Problems

The changes in Revision Two redesigned the application to make only one connect per update. The application still includes the following performance issues:

- The application is still serialized and chatty.
- The application still has a fat design; there are many sends that require no operation in this design.
- The sends are still only 3 bytes, which is poor streaming.

## Key Performance Metrics

The following performance metrics are expressed in Round Trip Time (RTT), Goodput, and Protocol Overhead. See the [Network Terminology](#) topic for an explanation of these terms.

This version reflects the following performance metrics:

- Cell Time — 1\*RTT
- Goodput — 4 bytes/RTT
- Protocol Overhead — 96.8%

## Related topics

[Improving a Slow Application](#)

Network Terminology

The Baseline Version: A Very Poor Performing Application

Revision 1: Cleaning up the Obvious

Revision 3: Compressed Block Send

Future Improvements

# Revision Three: Compressed Block Send

3/5/2021 • 2 minutes to read • [Edit Online](#)

In this version of the application, a compressed block send of the data is used. This change results in significant performance improvement.

```
BYTE tmp[3*ROWS*COLS];
FIELD Old = Map;
ComputeNext( Map );
n=Compact(Map,Old,tmp);
bind( s, ... );
connect( s, ... );
send( s, tmp, 3*n );
//can't do recv(s,tmp,n)
for(int i=0; i < n; )
    recv( s, tmp+i, n-i );
closesocket( s );
```

## Changes in this Version

This version reflects the following changes:

- Cell updates are no longer serialized.
- Since a block send is used, the application is no longer chatty.
- Data compression is used, resulting in a less fat application.

There is still an issue with this version of the application; the risk of deadlock exists since a large send is used with no receives. The server sends one byte for every 3 bytes received. This could cause a deadlock if the receive buffer size is less than 1000 bytes for this sample application.

## Key Performance Metrics

The following performance metrics are expressed in Round Trip Time (RTT), Goodput, and Protocol Overhead. See the [Network Terminology](#) topic for an explanation of these terms.

This version reflects the following performance metrics:

- Cell Time — .002\*RTT
- Goodput — 2 Kilobytes/RTT
- Protocol Overhead — 14%

Of the 14% overhead, 6% is from the Ethernet headers and the other 8% is from the connection startup and teardown.

## Related topics

[Improving a Slow Application](#)

[Network Terminology](#)

[The Baseline Version: A Very Poor Performing Application](#)

[Revision 1: Cleaning up the Obvious](#)

## Revision 2: Redesigning for Fewer Connects

### Future Improvements

# Future Improvements

3/5/2021 • 2 minutes to read • [Edit Online](#)

There are several improvements that can be made to this application, such as:

- A single, persistent connection could be created by the application. Appropriate error handling would have to be added. This would reduce the overhead associated with connection startup and teardown.
- The reply code on the server could be optimized to consolidate replies, reducing the number of packets sent from the server.
- Improvements in the protocol could be made. For example, an update bitmask could be used to signify which cells are to be updated, and only that cell data sent.
- Updates could be overlapped using different threads, so that the network is not idle while the `ComputeNext` function is running.

## Related topics

[Improving a Slow Application](#)

[The Baseline Version: A Very Poor Performing Application](#)

[Revision 1: Cleaning up the Obvious](#)

[Revision 2: Redesigning for Fewer Connects](#)

[Revision 3: Compressed Block Send](#)

# Best Practices for Interactive Applications

3/5/2021 • 2 minutes to read • [Edit Online](#)

In morphing the Life cell update code, several guidelines for writing high performance network applications have been uncovered. Some general strategies to apply when writing these types of applications are:

- Make the data stream as much as possible, rather than going in chunks.
- Use a few large transactions rather than many small ones. Large transactions can also be efficiently streamed.
- Recognize that the network is a slow, unreliable resource and develop each application to minimize its reliance on the network.
- Use a well-architected representation of the data on the network. The data representation should be computer-architecture agnostic, contain no fat, and possibly be compressed.
- During initialization and shutdown, do not make the user wait for the network to start up or shut down. Network related initialization could take a relatively long time. Separate the noncritical network code.
- Handle errors as appropriate to their impact. Not all errors are critical. Implement recovery mechanisms and provide nonintrusive user feedback.
- Use remote procedure calls (RPC) only when appropriate. RPC is synchronous on Windows Me/98 and always results in chatty, fat protocols when used to send small amounts of data.
- Measure your network overhead using Netstat; you may be surprised at what your measurements reveal.
- Test the application on a variety of networks, especially slow or loss-prone networks. Wireless LAN networks, modems, and virtual private networks (VPN) over the Internet are good networks for testing.

## Related topics

[High-performance Windows Sockets Applications](#)

# Categorizing layered service providers and apps

3/5/2021 • 8 minutes to read • [Edit Online](#)

## NOTE

Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use [Windows Filtering Platform](#).

Winsock 2 accommodates layered protocols. 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 a layered protocol or layered service provider would be a security layer that adds protocol to the connection establishment process in order to perform authentication and to establish a mutually agreed upon encryption scheme. Such a security protocol would generally require the services of an underlying reliable transport protocol such as TCP or SPX. The term base protocol implemented by base provider refers to a Winsock provider that implements a protocol such as TCP or SPX which is capable of performing data communications with a remote endpoint. The term layered protocol is used to describe a protocol that cannot stand alone. These layered protocols are installed as Winsock Layered Service Providers (LSPs).

An example of an LSP is the Microsoft Firewall Client Service Provider installed as part of the Internet Security and Authentication Server (ISA) on clients. The Microsoft Firewall Client Service Provider installs over the Winsock base providers for TCP and UDP. A dynamic-link library (DLL) in the ISA Firewall Client software becomes a Winsock layered service provider that all Winsock applications use transparently. This way, the ISA Firewall Client LSP can intercept Winsock function calls from client applications and then route a request to the original underlying base service provider if the destination is local or to the Firewall service on an ISA Server computer if the destination is remote. A similar LSP is installed as part of the Microsoft Forefront Firewall Service and the Threat Management Gateway (TMG) Client on clients.

During LSP initialization, the LSP must provide pointers to a number of Winsock Service Provider Interface (SPI) functions. These functions will be called during normal processing by the layer directly above the LSP (either another LSP or Ws2\_32.DLL).

It is possible to define LSP categories based upon the subset of SPI functions an LSP implements and the nature of the extra processing performed for each of those functions. By classifying LSPs, as well as classifying applications which use Winsock sockets, it becomes possible to selectively determine if an LSP should be involved in a given process at runtime.

On Windows Vista and later, a new method is provided for categorizing both Winsock Layered Service Providers and applications so that only certain LSPs will be loaded. There are several reasons for adding these features.

One of the primary reasons is that certain system critical processes such as winlogon and lsass create sockets, but these processes do not use these sockets to send any traffic on the network. So most LSPs should not be loaded into these processes. A number of cases have also been documented where buggy LSPs can cause *lsass.exe* to crash. If lsass crashes, the system forces a shutdown. A side affect of these system processes loading LSPs is that these processes never exit so when an LSP is installed or removed, a reboot is required.

A secondary reason is that there are some cases where applications may not want to load certain LSPs. For example, some applications may not want to load cryptographic LSPs which could prevent the application from communicating with other systems that do not have the cryptographic LSP installed.

Finally, LSP categories can be used by other LSPs to determine where in the Winsock protocol chain they should install themselves. For years, various LSP developers have wanted a way of knowing how an LSP will behave. For example, an LSP that inspects the data stream would want to be above an LSP that encrypts the data. Of course, this method of LSP categorization isn't fool proof since it relies on third-party LSPs to categorize themselves appropriately.

The LSP categorization and other security enhancements in Windows Vista and later are designed to help prevent users from unintentionally installing malicious LSPs.

## LSP Categories

On Windows Vista and later, an LSP can be classified based on how it interacts with Windows Sockets calls and data. An LSP category is an identifiable group of behaviors on a subset of Winsock SPI functions. For example, an HTTP content filter would be categorized as a data inspector (the **LSP\_INSPECTOR** category). The **LSP\_INSPECTOR** category will inspect (but not alter) parameters to data transfer SPI functions. An application can query for the category of an LSP and choose to not load the LSP based on the LSP category and the application's set of permitted LSP categories.

The following table lists categories that an LSP can be classified into.

LSP CATEGORY	DESCRIPTION
<b>LSP_CRYPTO_COMPRESS</b>	The LSP is a cryptography or data compression provider.
<b>LSP_FIREWALL</b>	The LSP is a firewall provider.
<b>LSP_LOCAL_CACHE</b>	The LSP is a local cache provider.
<b>LSP_INBOUND MODIFY</b>	The LSP modifies inbound data.
<b>LSP_INSPECTOR</b>	The LSP inspects or filters data.
<b>LSP_OUTBOUND MODIFY</b>	The LSP modifies outbound data.
<b>LSP_PROXY</b>	The LSP acts as a proxy and redirects packets.
<b>LSP_REDIRECTOR</b>	The LSP is a network redirector.
<b>LSP_SYSTEM</b>	The LSP is acceptable for use in services and system processes.

An LSP may belong to more than one category. For example, a firewall/security LSP could belong to both the inspector (**LSP\_INSPECTOR**) and firewall (**LSP\_FIREWALL**) categories.

If an LSP does not have a category set, it is considered to be in the All Other category. This LSP category will not be loaded in services or system processes (for example, lsass, winlogon, and many svchost processes).

## Categorizing LSPs

Several new functions are available on Windows Vista and later for categorizing an LSP:

- [WSCGetProviderInfo](#)
- [WSCGetProviderInfo32](#)

- [WSCSetProviderInfo](#)
- [WSCSetProviderInfo32](#)

In order to categorize an LSP, the [WSCSetProviderInfo](#) or [WSCSetProviderInfo32](#) function is called with a GUID to identify the LSP hidden entry, the information class to be set for this LSP protocol entry, and a set of flags used to modify the behavior of the function.

The [WSCGetProviderInfo](#) or [WSCGetProviderInfo32](#) function is similarly used to retrieve the data associated with an information class for an LSP.

## Categorizing Applications

Several new functions are available on Windows Vista and later for categorizing an application:

- [WSCGetApplicationCategory](#)
- [WSCSetApplicationCategory](#)

In order to categorize an application, the [WSCSetApplicationCategory](#) function is called with the load path to the executable image to identify the application, the command line arguments used when starting the application, and the LSP categories which are permitted for all instances of this application.

The [WSCGetApplicationCategory](#) function is similarly used to retrieve the layered service provider (LSP) categories associated with an application.

## Determining Which LSPs Get Loaded

The final part of LSP categorization is determining which LSPs will be loaded into which processes. When a process loads Winsock, the following comparisons are made of the application category and the LSP categories for all installed LSPs:

- If the application is not categorized, allow all LSPs to be loaded into the process.
- If both the application and the LSP have assigned categories, all of the following must be true:

At least one of the LSP categories is present in the application's specified categories.

Only categories specified in the application's specified categories are specified in the LSPs categories. For example, if the application specifies a category it must be in the LSP's category.

If the LSP\_SYSTEM category is present in the application's category, it must be present in the LSP's categories.

### NOTE

If an LSP is not categorized, its category is effectively zero. For a match to occur, all the LSP's specified categories must be present in the application's categories (the application's categories must be a superset of the LSP's categories) with the caveat that if LSP\_SYSTEM is present in the application's category it must also be present in the LSP's category.

Consider the following example:

The *Foo.exe* application is categorized as LSP\_SYSTEM + LSP\_FIREWALL + LSP\_CRYPTO\_COMPRESS. The application *Bar.exe* is categorized as LSP\_FIREWALL + LSP\_CRYPTO\_COMPRESS. There are four LSPs installed on the system:

- LSP1 has set a category of LSP\_SYSTEM.
- LSP2 is not categories set, so its category is zero.
- LSP3 has set a category of LSP\_FIREWALL.

- LSP4 has set categories of LSP\_SYSTEM + LSP\_FIREWALL + LSP\_CRYPTO\_COMPRESS + LSP\_INSPECTOR

In this example, the *Foo.exe* application would only load LSP1, while the *Bar.exe* application would load LSP3.

## Determining Winsock Providers Installed

The Microsoft Windows Software Development Kit (SDK) includes a sample Winsock program that can be used to determine the Winsock transport providers installed on a local computer. By default, the source code for this Winsock sample is installed in the following directory of the Windows SDK for Windows 7:

*C:\Program Files\Microsoft SDKs\Windows\v7.0\Samples\NetDs\winsock\LSP*

This sample is a utility for installing and testing layered service providers. But it can also be used to programmatically collect detailed information from the Winsock catalog on a local computer. To list all of the current Winsock providers including both base providers and layer service providers, build this Winsock sample and execute the following console command:

**instlsp -p**

The output will be a list of Winsock providers installed on the local computer including layered service providers. The output lists the catalog ID and the string name for the Winsock provider

To collect more detailed information on all Winsock providers, execute the following console command:

**instlsp -p -v**

The output will be a list **WSAPROTOCOL\_INFO** structures supported on the local computer.

For a list of only layered service providers installed on the local computer, execute the following console command:

**instlsp -l**

To map the LSP structure, execute the following console command:

**instlsp -m**

### NOTE

The TDI feature is deprecated and will be removed in future versions of Microsoft Windows. Depending on how you use TDI, use either the Winsock Kernel (WSK) or Windows Filtering Platform (WFP). For more information about WFP and WSK, see [Windows Filtering Platform](#) and [Winsock Kernel](#). For a Windows Core Networking blog entry about WSK and TDI, see [Introduction to Winsock Kernel \(WSK\)](#).

# Multicast Programming

3/5/2021 • 3 minutes to read • [Edit Online](#)

Multicast programming is enabled through Windows Sockets. Windows Sockets enables the Multicast Listener Discovery (MLD) versions 1 (MLDv1) and 2 (MLDv2) on IPv6 and the Internet Group Management Protocol versions 2 (IGMPv2) and 3 (IGMPv3) through the use of socket options or IOCTLs. This section describes the Windows implementation, explains how to enable multicast programming using Windows Sockets, and provides programming samples to illustrate its use.

The second version of IGMP, hereafter referred to as IGMPv2, enables hosts to join and leave multicast groups identified by an IPv4 multicast address on a specific network interface. Windows Sockets enables an application to join and leave such groups on specific sockets. A drawback of IGMPv2, however, is that any IPv4 source address joined to the IGMPv2 group can transmit to all members, potentially flooding the group and making it unusable for transmissions that require a primary source, such as an Internet radio station. The problem with IGMPv2 is its inability to selectively choose a single IPv4 source address (or even a few sources), and its inability to block senders (such as rogue broadcasters or denial-of-service perpetrators) for a given multicast group. IGMPv3 addresses those shortcomings.

With Windows Sockets and IGMPv3, applications can select a particular multicast IPv4 source address and multicast group pair. In addition, Windows Sockets enables developers to selectively allow additional broadcasters in a given source/group pair, or enables applications to block specific broadcasters. IGMPv3 is supported on Windows Vista and later.

The first version of MLD on IPv6, referred to as MLDv1, is very similar to IGMPv2 and suffers from the same limitations. MLDv1 enables hosts to join and leave multicast groups identified by an IPv6 multicast address on a specific network interface. Windows Sockets enables an application to join and leave such groups on specific sockets. However, any IPv6 source address joined to the MLDv1 group can transmit to all members, potentially flooding the group and making it unusable for transmissions that require a primary source. The problem with MLDv1 is its inability to selectively choose a single IPv6 source address (or even a few sources), and its inability to block senders (such as rogue broadcasters or denial-of-service perpetrators) for a given multicast group. MLDv2 addresses those shortcomings.

With Windows Sockets and MLDv2, applications can select a particular multicast IPv6 source address and multicast group pair. In addition, Windows Sockets enables developers to selectively allow additional broadcasters in a given source/group pair, or enables applications to block specific broadcasters. MLDv2 is supported on Windows Vista and later.

There are two approaches an application programmer can take when developing multicast applications in Windows. The first approach is change-based; multicast sources are added or removed using socket options, even during the course of transmission, as required. The second approach is final-state based; source addresses and any included/excluded addresses are specified with an IOCTL. Each approach is a valid multicasting practice, but developers may find using socket options and the change-based approach more intuitive and flexible.

This section has the following pages:

PAGE TITLE	DESCRIPTION
<a href="#">MLD and IGMP Using Windows Sockets</a>	Enumerates the multicast socket options available for use in Windows Sockets programming, using a change-based programming approach. Defines two multicast application categories.

PAGE TITLE	DESCRIPTION
<a href="#">Multicast Socket Option Behavior</a>	Provides an extensive table to explain the implications and requirements of calling multicast socket options in particular order.
<a href="#">Multicast Programming Sample</a>	Programming snippet that illustrates how to use socket options to enable multicast applications in Windows.
<a href="#">Final-State-based Multicast Programming</a>	Explains final-state approach, and how to use IOCTLs for multicast programming with Windows Sockets.
<a href="#">Porting Broadcast Applications to IPv6</a>	Provides guidelines for porting IPv4 broadcast applications to IPv6 multicast.

# MLD and IGMP Using Windows Sockets

3/5/2021 • 3 minutes to read • [Edit Online](#)

Windows Sockets enables the Multicast Listener Discovery (MLD) on IPv6 and the Internet Group Management Protocol (IGMP) on IPv4 for multicast applications through the use of socket options and IOCTLs. This page describes the socket options that enable multicast programming, and describes how they are used. For definitions of each socket option, consult the [Socket Options](#) page.

For information on using IOCTLs for multicast programming, see [Final-State-Based Multicast Programming](#) later in this section.

On Windows Vista and later, a set of socket options are available for multicast programming that support IPv6 and IPv4 addresses. These socket options are IP agnostic and can be used on both IPv6 and IPv4. On IPv6, these socket options use MLDv2. On IPv4, these socket options use IGMPv3. These IP agnostic options are the preferred socket options for multicast programming on Windows Vista and later. Windows Sockets uses the following socket options:

SOCKET OPTION	ARGUMENT TYPE
MCAST_BLOCK_SOURCE	<a href="#">GROUP_SOURCE_REQ</a> structure
MCAST_JOIN_GROUP	<a href="#">GROUP_REQ</a> structure
MCAST_JOIN_SOURCE_GROUP	<a href="#">GROUP_SOURCE_REQ</a> structure
MCAST_LEAVE_GROUP	<a href="#">GROUP_REQ</a> structure
MCAST_LEAVE_SOURCE_GROUP	<a href="#">GROUP_SOURCE_REQ</a> structure
MCAST_UNBLOCK_SOURCE	<a href="#">GROUP_SOURCE_REQ</a> structure

A set of socket options are available for multicast programming that support IPv6 only addresses. These socket options use MLDv1 or MLDv2. The version of MLD used is dependent on the version of Windows. MLDv2 is supported on Windows Vista and later. Windows Sockets uses the following socket options:

SOCKET OPTION	ARGUMENT TYPE
IPV6_ADD_MEMBERSHIP	<a href="#">ipv6_mreq</a> structure
IPV6_DROP_MEMBERSHIP	<a href="#">ipv6_mreq</a> structure

A set of socket options are available for multicast programming that support IPv4 only addresses. These socket options use IGMPv3 or IGMPv2. The version of IGMP used is dependent on the version of Windows. IGMPv3 is supported on Windows Vista and later. Windows Sockets uses the following socket options:

SOCKET OPTION	ARGUMENT TYPE
IP_ADD_MEMBERSHIP	<a href="#">ip_mreq</a> structure
IP_ADD_SOURCE_MEMBERSHIP	<a href="#">ip_mreq_source</a> structure
IP_BLOCK_SOURCE	<a href="#">ip_mreq_source</a> structure
IP_DROP_MEMBERSHIP	<a href="#">ip_mreq</a> structure
IP_DROP_SOURCE_MEMBERSHIP	<a href="#">ip_mreq_source</a> structure
IP_UNBLOCK_SOURCE	<a href="#">ip_mreq_source</a> structure

When IGMPv3 is available, the IP\_ADD\_SOURCE\_MEMBERSHIP, IP\_BLOCK\_SOURCE, IP\_DROP\_SOURCE\_MEMBERSHIP, and IP\_UNBLOCK\_SOURCE options are handled more efficiently since the router can handle the filtering. When only IGMPv2 is available, the host must handle the filtering.

There are two categories into which most applications are likely to fall: any-source and controlled-source.

- **Any-source** applications accept all sources by default, allowing individual sources to be turned off as required. An example of an any-source application is a video conference call that enables all recipients to participate.
- **Controlled-source** applications limit sources to a given list, such as an Internet radio station, or the broadcast of a notable event. The process of using socket options is slightly different for each.

On Windows Vista and later, the following steps apply for any-source applications:

- Use **MCAST\_JOIN\_GROUP** to join a group.
- Use **MCAST\_BLOCK\_SOURCE** to turn off a given source, if required.
- Use **MCAST\_UNBLOCK\_SOURCE** to re-allow a blocked source, if required.
- Use **MCAST\_LEAVE\_GROUP** to leave the group.

On Windows Vista and later, the following steps apply for controlled-source applications:

- Use **MCAST\_JOIN\_SOURCE\_GROUP** to join each group/source pair.
- Use **MCAST\_LEAVE\_SOURCE\_GROUP** to leave each group/source, or use **MCAST\_LEAVE\_GROUP** to leave all sources, if the same group address is used by all sources.

The following steps apply for any-source applications:

- Use **IP\_ADD\_MEMBERSHIP** to join a group (**IPV6\_ADD\_MEMBERSHIP** for IPv6).
- Use **IP\_BLOCK\_SOURCE** to turn off a given source, if required.
- Use **IP\_UNBLOCK\_SOURCE** to re-allow a blocked source, if required.
- Use **IP\_DROP\_MEMBERSHIP** to leave the group (**IPV6\_DROP\_MEMBERSHIP** for IPv6).

The following steps apply for controlled-source applications:

- Use **IP\_ADD\_SOURCE\_MEMBERSHIP** to join each group/source pair.
- Use **IP\_DROP\_SOURCE\_MEMBERSHIP** to leave each group/source, or use **IP\_DROP\_MEMBERSHIP** to leave all sources, if the same group address is used by all sources.

The order in which these socket options are set has associated rules. For information and troubleshooting information on multicast socket options, see [Multicast Socket Option Behavior](#).

# Multicast Socket Option Behavior

3/5/2021 • 2 minutes to read • [Edit Online](#)

This page describes the behavior of multicast socket options based on various socket option settings states.

For example, this page describes the behavior when the IP\_ADD\_SOURCE\_MEMBERSHIP socket option is set on a socket for which the IP\_ADD\_SOURCE\_MEMBERSHIP option has already been set with the specified group/source pair on the same network interface. It is permitted to call IP\_ADD\_SOURCE\_MEMBERSHIP on the same group on a different network interface.

This page assists in properly designing and troubleshooting Windows Sockets multicast applications.

INITIAL SOCKET OPTION	CONFLICTING SUBSEQUENT SOCKET OPTION	ERROR RETURNED	REMARKS
IP_ADD_MEMBERSHIP\${REMOVE}\$	IP_ADD_MEMBERSHIP	WSAEADDRNOTAVAIL	Do not call IP_ADD_MEMBERSHIP with the same group more than once on the same network interface.
	IP_ADD_SOURCE_MEMBERSHIP	WSAEADDRNOTAVAIL	Do not call IP_ADD_SOURCE_MEMBERSHIP with the same group previously called with IP_ADD_MEMBERSHIP on the same network interface.
	IP_DROP_SOURCE_MEMBERSHIP	WSAEINVAL	Use IP_BLOCK_SOURCE instead.
	IP_UNBLOCK_SOURCE	WSAEINVAL	Returns an error when attempting to unblock a group/source pair that has not previously been blocked on the same network interface.
IP_DROP_MEMBERSHIP	Any subsequent call on the same group or group/source pair	WSAEINVAL	Making socket option calls on a group or group/source pair not currently in the inclusion list (due to dropping membership, or otherwise) results in an error.
IP_ADD_SOURCE_MEMBERSHIP\${REMOVE}\$	IP_ADD_MEMBERSHIP	WSAEADDRNOTAVAIL	Do not call IP_ADD_MEMBERSHIP with the same group previously called with IP_ADD_SOURCE_MEMBERSHIP on the same network interface.

INITIAL SOCKET OPTION	CONFLICTING SUBSEQUENT SOCKET OPTION	ERROR RETURNED	REMARKS
	IP_ADD_SOURCE_MEMBERSHIP	WSAEADDRNOTAVAIL	Do not call IP_ADD_SOURCE_MEMBERSHIP with the same group/source pair previously called with IP_ADD_SOURCE_MEMBERSHIP on the same network interface.
	IP_UNBLOCK_SOURCE	WSAEINVAL	Returns an error when attempting to unblock a group/source pair that has not previously been blocked on the same network interface.
IP_DROP_SOURCE_MEMBERSHIP\${REMOVE}\$	IP_UNBLOCK_SOURCE	WSAEINVAL	Returns an error when attempting to unblock a group/source pair that has not previously been blocked on the same network interface.
	IP_DROP_SOURCE_MEMBERSHIP	WSAEADDRNOTAVAIL	Returns an error when attempting to drop a group/source pair that is not in the inclusion list on the same network interface.
IP_BLOCK_SOURCE\${REMOVE}\$	IP_BLOCK_SOURCE	WSAEADDRNOTAVAIL	Returns an error when attempting to block a group/source pair that is already blocked on the same network interface.
	IP_ADD_SOURCE_MEMBERSHIP	WSAEINVAL	Use IP_UNBLOCK_SOURCE instead.
	IP_ADD_MEMBERSHIP	WSAEINVAL	Use IP_UNBLOCK_SOURCE instead.
IP_UNBLOCK_SOURCE	IP_UNBLOCK_SOURCE	WSAEADDRNOTAVAIL	Returns an error when attempting to unblock a group/source pair that is not in the blocked list on the same network interface.

# Multicast Programming Sample

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following sample code illustrates how to include multicast functionality to a Windows Sockets application using socket options.

```
#define WIN32_LEAN_AND_MEAN

#include <winsock2.h>
#include <Ws2tcpip.h>
#include <mswsock.h>

#define u_int32 UINT32 // Unix uses u_int32

// Need to link with Ws2_32.lib
#pragma comment (lib, "Ws2_32.lib")

int /* OUT: whatever setsockopt() returns */
join_source_group(int sd, u_int32 grpaddr,
    u_int32 srcaddr, u_int32 iaddr)
{
    struct ip_mreq_source imr;

    imr.imr_multiaddr.s_addr = grpaddr;
    imr.imr_sourceaddr.s_addr = srcaddr;
    imr.imr_interface.s_addr = iaddr;
    return setsockopt(sd, IPPROTO_IP, IP_ADD_SOURCE_MEMBERSHIP, (char *) &imr, sizeof(imr));
}

int
leave_source_group(int sd, u_int32 grpaddr,
    u_int32 srcaddr, u_int32 iaddr)
{
    struct ip_mreq_source imr;

    imr.imr_multiaddr.s_addr = grpaddr;
    imr.imr_sourceaddr.s_addr = srcaddr;
    imr.imr_interface.s_addr = iaddr;
    return setsockopt(sd, IPPROTO_IP, IP_DROP_SOURCE_MEMBERSHIP, (char *) &imr, sizeof(imr));
}
```

# Final-State-Based Multicast Programming

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section describes final-state-based multicast programming using IOCTLs instead of socket options. For an overview of how final-state-based multicast programming differs from change-based multicast programming, see [Multicast Programming](#).

The following table describes the Windows Sockets IOCTLs used for multicast programming on Windows.

IOCTL	ARGUMENT TYPE
SIOCSMSFILTER	<a href="#">GROUP_FILTER</a> structure
SIOCGMSFILTER	<a href="#">GROUP_FILTER</a> structure
SIO_GET_MULTICAST_FILTER	<a href="#">ip_msfilter</a> structure
SIO_SET_MULTICAST_FILTER	<a href="#">ip_msfilter</a> structure

Note that the **SIOCSMSFILTER** and **SIOCGMSFILTER** IOCTLs are available on Windows Vista and later.

Using these IOCTLs for multicast programming has performance benefits when working with large source lists. For more information about the parameters and settings associated with using SIO\_GET\_MULTICAST\_FILTER or SIO\_SET\_MULTICAST\_FILTER, consult the [GROUP\\_FILTER](#) reference page. For more information about the parameters and settings associated with using SIO\_GET\_MULTICAST\_FILTER or SIO\_SET\_MULTICAST\_FILTER, consult the [ip\\_msfilter](#) reference page.

# Porting Broadcast Applications to IPv6

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section describes best practices for porting an IPv6 broadcast application to the multicast capabilities available with Windows Sockets.

## Comparing IPv4 to IPv6

The most notable consideration when preparing to port an IPv4 broadcast application to IPv6 is this: IPv6 has no implemented concept of broadcast. Instead, IPv6 uses multicast.

Multicast for IPv6 can emulate traditional broadcast capabilities found in IPv4. Setting the [IPV6\\_ADD\\_MEMBERSHIP](#) socket option with the IPv6 address set to the link-local scope all nodes address (FF02::1) is equivalent to broadcasting on IPv4 broadcast addresses using the [SO\\_BROADCAST](#) socket option. This address is sometimes called the *all-nodes multicast group*. For applications that simply want broadcast emulation for IPv6, that approach is operationally equivalent. One notable difference with IPv6, however, is that multicasts on the all-nodes multicast group address are not received by default (IPv4 broadcasts are received by default). Application programmers must use the [IPV6\\_ADD\\_MEMBERSHIP](#) socket option to enable multicast reception from any source, including the all-nodes multicast group address.

### NOTE

In IPv6, interface selection is specified in the argument structure passed to the multicast socket option or IOCTL.

But for richer, more robust, more selective and more manageable transmissions to multiple hosts, application developers should consider moving to a multicast model.

## Moving to Multicast

With multicast, application programmers can selectively choose a particular source/group pair, enabling a selective reception model. The Multicast Listener Discovery (MLD) on IPv6 and version 3 of the Internet Group Management Protocol (IGMPv3) on IPv4 support multicasting programming. In addition, multicast enables applications to specifically block (or unblock) senders within a group, further protecting applications from rogue broadcasters. This capability is available for IPv4 (requires IGMPv3) as well as IPv6 (requires MLDv2).

There are two primary scenarios for application programmers using multicast: those porting from IPv4 broadcast (or multicast) applications to IPv6, and those creating new IPv6 multicast applications.

For porting existing applications, there are two options to move to IPv6 multicast: using socket options and using IOCTLs.

- Using socket options is a change-based approach, which enables developers to change the socket properties as required (such as blocking or unblocking a sender, adding a new source, and so on). This approach is more intuitive and the recommended approach. For more information on the change-based approach to multicast programming, see [MLD and IGMP Using Windows Sockets](#).
- Using IOCTLs is a final-state based approach, because it enables developers to provide a fully-configured socket state, including inclusion and exclusion lists, with one call. For more information on the final-state based approach, see [Final-State-Based Multicast Programming](#).

For those creating new IPv6 multicast applications, the recommended practice is to use socket options, rather than using IOCTLs.

There is one other approach to creating multicast applications with IPv6, and that entails using the **WSAJoinLeaf** function. While using the **WSAJoinLeaf** function is not the recommended practice, there are situations that may dictate its use. For example, one drawback to using socket options on Windows Server 2003 and earlier is that they are IP version specific. On these older versions of Windows, different sockets options must be for IPv6 and IPv4. On Windows Vista and later, new socket options are supported that can be used with both IPv4 and IPv6. The **WSAJoinLeaf** function, in contrast, is IP version and protocol agnostic, and therefore it can be a useful approach for building an application that must work with multiple IP versions on Windows Server 2003 and earlier. Using the **WSAJoinLeaf** function may be more appropriate in certain situations where protocol and IP-version agnosticism is required.

# Reliable Multicast Programming (PGM)

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section describes the Pragmatic General Multicast (PGM) multicast protocol implementation in Windows, often referred to as reliable multicast. Reliable multicast is implemented through Windows Sockets in Windows Server 2003 and later.

**Windows XP:** PGM is only supported when Microsoft Message Queuing (MSMQ) 3.0 is installed.

PGM is a reliable and scalable multicast protocol that enables receivers to detect loss, request retransmission of lost data, or notify an application of unrecoverable loss. PGM is a receiver-reliable protocol, which means the receiver is responsible for ensuring all data is received, absolving the sender of reception responsibility.

PGM is appropriate for applications that require duplicate-free multicast data delivery from multiple sources to multiple receivers. PGM does not support acknowledged delivery, nor does it guarantee ordering of packets from multiple senders.

For more information about PGM, refer to RFC 3208 available at [www.ietf.org](http://www.ietf.org).

This section describes how to use reliable multicast on Windows. The following topics explain the various aspects of creating a reliable multicast application using Windows Sockets:

- [PGM Senders and Receivers](#)
- [PGM Sender Options](#)
- [Sending and Receiving PGM Data](#)
- [Multihoming and PGM](#)
- [PGM Socket Options](#)

# PGM Senders and Receivers

3/5/2021 • 4 minutes to read • [Edit Online](#)

Establishing a PGM session is similar to the connection establishment routine associated with a TCP session. The significant departure from a TCP session, however, is that the client and server semantics are reversed; the server (the PGM sender) connects to a multicast group, while the client (the PGM receiver) waits to accept a connection. The following paragraphs detail programmatic steps necessary for creating a PGM sender and a PGM receiver. This page also describes the available data modes for PGM sessions.

## PGM Sender

To create a PGM sender, perform the following steps

1. Create a PGM socket.
2. **bind** the socket to INADDR\_ANY.
3. **connect** to the multicast group transmission address.

No formal session handshaking is performed with any clients. The connection process is similar to a UDP **connect**, in that it associates an endpoint address (the multicast group) with the socket. Once completed, data may be sent on the socket.

When a sender creates a PGM socket and connects it to a multicast address, a PGM session is created. A reliable multicast session is defined by a combination of the globally unique identifier (GUID) and the source port. The GUID is generated by the transport. The sSource port is specified by the transport, and no control is provided over which source port is used.

### NOTE

Receiving data on a sender socket is not allowed, and results in an error.

The following code snippet illustrates setting up a PGM sender:

```

SOCKET      s;
SOCKADDR_IN salocal, sasession;
int         dwSessionPort;

s = socket (AF_INET, SOCK_RDM, IPPROTO_RM);

salocal.sin_family = AF_INET;
salocal.sin_port   = htons (0);    // Port is ignored here
salocal.sin_addr.s_addr = htonl (INADDR_ANY);

bind (s, (SOCKADDR *)&salocal, sizeof(salocal));

//
// Set all relevant sender socket options here
//

//
// Now, connect <entity type="hellip"/>
// Setting the connection port (dwSessionPort) has relevance, and
// can be used to multiplex multiple sessions to the same
// multicast group address over different ports
//
dwSessionPort = 0;
sasession.sin_family = AF_INET;
sasession.sin_port   = htons (dwSessionPort);
sasession.sin_addr.s_addr = inet_addr ("234.5.6.7");

connect (s, (SOCKADDR *)&sasession, sizeof(sasession));

//
// We're now ready to send data!
//

```

## PGM Receiver

To create a PGM receiver, perform the following steps

1. Create a PGM socket.
2. **bind** the socket to the multicast group address on which the sender is transmitting.
3. Call the **listen** function on the socket to put the socket in listening mode. The listen function returns when a PGM session is detected on the specified multicast group address and port.
4. Call the **accept** function to obtain a new socket handle corresponding to the session.

Only original PGM data (ODATA) triggers the acceptance of a new session. As such, other PGM traffic (such as SPM or RDATA packets) may be received by the transport, but do not result in the **listen** function returning.

Once a session is accepted, the returned socket handle is used for receiving data.

### NOTE

Sending data on a receive socket is not allowed, and results in an error.

The following code snippet illustrates setting up a PGM receiver:

```

SOCKET      s,
            sclient;
SOCKADDR_IN salocal,
            sasession;
int         sasessionsz, dwSessionPort;

s = socket (AF_INET, SOCK_RDM, IPPROTO_RM);

//
// The bind port (dwSessionPort) specified should match that
// which the sender specified in the connect call
//
dwSessionPort = 0;
salocal.sin_family = AF_INET;
salocal.sin_port   = htons (dwSessionPort);
salocal.sin_addr.s_addr = inet_addr ("234.5.6.7");

bind (s, (SOCKADDR *)&salocal, sizeof(salocal));

//
// Set all relevant receiver socket options here
//

listen (s, 10);

sasessionsz = sizeof(sasession);
sclient = accept (s, (SOCKADDR *)&sasession, &sasessionsz);

//
// accept will return the client socket and we are now ready
// to receive data on the new socket!
//

```

## Data Modes

PGM sessions have two options for data modes: message mode and stream mode.

Message mode is appropriate for applications that need to send discrete messages, and is specified by a socket type of SOCK\_RDM. Stream mode is appropriate for applications that need to send streaming data to receivers, such as video or voice applications, and is specified by a socket type of SOCK\_STREAM. The choice of mode effects how Winsock processes data.

Consider the following example: A message mode PGM sender makes three calls to the [WSASend](#) function, each with a 100-byte buffer. This operation appears on the wire as three discrete PGM packets. On the receiver side, each call to the [WSARecv](#) function returns only 100 bytes, even if a larger receive buffer is provided. In contrast, with a stream mode PGM sender those three 100 byte transmissions could be coalesced into less than three physical packets on the wire (or coalesced into one blob of data on the receiver side). As such, when the receiver calls one of the Windows Sockets receive functions, any amount of data that has been received by the PGM transport may be returned to the application without regard to how the data was physically transmitted or received.

# PGM Sender Options

3/5/2021 • 2 minutes to read • [Edit Online](#)

PGM senders are provided with certain default settings that affect the performance of data transmission, and how long data is buffered to account for packet loss and associated PGM client retransmission requests. The following paragraphs describe these default settings.

## Window Size and Transmission Rate

The capability to set window size and transmission rate enables applications to control the amount of data the transport buffers for retransmission, and the rate at which the byte-stream is transmitted.

Retransmission data is stored in a file, therefore the maximum window size is limited by disk space usable by the transport. The default window size is 10MB. Although it is possible for a send or message size to exceed the window or buffer size, the data stream remain uninterrupted; the send is pended until the all the data has been sent out.

### NOTE

The maximum buffer space is limited by the maximum number of packets that can be held in the window at any given time, which is equal to  $2^{31} - 1$ .

The transmission rate is the combined outflow of original data packets (ODATA), retransmitted data packets (RDATA) and transport-specific bookkeeping packets (SPMs), expressed per second. If the rate limit is set to 56 kilobits per sec by default. The default window size is 10 megabytes, with a default rate of 56 kilobits per second. Due to the relationship between the three members of the [RM\\_SEND\\_WINDOW](#) structure, the default window size is therefore 1428 seconds. See [RM\\_SEND\\_WINDOW](#) for more information.

## Window Advance Rate

The window advance rate is set by the [RM\\_SENDER\\_WINDOW\\_ADV\\_RATE](#) socket option. This option enables applications to specify the increment at which the PGM sender's window is advanced, expressed as a nonzero percentage value of the window size. The default value is 15%, and the maximum rate is 50%. If the PGM sender has repair data pending that falls in the space of the increment window, the window is advanced partially as each repair packet in the window is sent out.

## Forward Error Correction (FEC)

Forward error correction is set through use of the [RM\\_USE\\_FEC](#) socket option. This socket option enables the PGM sender to send repair packets as parity packets instead of regular data packets. Doing so minimizes the number of repair packets sent to repair different sequences lost by multiple receivers from within the same data group. Enabling FEC is only set on the PGM sender. PGM receivers automatically follow the policy set by the sender. For a detailed discussion on FEC, refer to the PGM RFC located on the [IETF](#) website.

# Sending and Receiving PGM Data

3/5/2021 • 3 minutes to read • [Edit Online](#)

Sending and receiving PGM data is similar to sending or receiving data on any socket. There are considerations specific to PGM, outlined in the following paragraphs.

## Sending PGM Data

Once a PGM sender session is created, data is sent using the various Windows Sockets send functions: [send](#), [sendto](#), [WSASend](#), and [WSASendTo](#). Since Windows Sockets handles are file system handles, other functions such as [WriteFile](#) and CRT functions can also transmit data. The following code snippet illustrates a PGM sender operation:

```
LONG      error;
//:
error = send (s, pSendBuffer, SendLength, 0);
if (error == SOCKET_ERROR)
{
    fprintf (stderr, "send() failed: Error = %d\n",
             WSAGetLastError());
}
```

When using message mode (SOCK\_RDM), each call to a send function results in a discrete message, which sometimes isn't desirable; an application may want to send a 2 megabyte message with multiple calls to [send](#). In such circumstances, the sender can set the [RM\\_SET\\_MESSAGE\\_BOUNDARY](#) socket option to indicate the size of the message that follows.

If the send window is full, a new send from the application is not accepted until window has been advanced. Attempting to send on a non-blocking socket fails with [WSAEWOULDBLOCK](#); a blocking socket simply blocks until the window advances to the point where the requested data can be buffered and sent. In overlapped I/O, the operation does not complete until the window advances enough to accommodate the new data.

## Receiving PGM Data

Once a PGM receiver session is created, data is received using the various Windows Sockets receive functions: [recv](#), [recvfrom](#), [WSARecv](#), and [WSARecvFrom](#). Since Windows Sockets handles are also file handles, the [ReadFile](#) and CRT functions can also be used to receive PGM session data. The transport forwards data up to the receiver as it arrives as long as the data is in sequence. The transport guarantees that the data returned is contiguous and free of duplicates. The following code snippet illustrates a PGM receive operation:

```
LONG      BytesRead;
//:
BytesRead = recv (sockR, pTestBuffer, MaxBufferSize, 0);
if (BytesRead == 0)
{
    fprintf(stdout, "Session was terminated\n");
}
else if (BytesRead == SOCKET_ERROR)
{
    fprintf(stderr, "recv() failed: Error = %d\n",
            WSAGetLastError());
}
```

When using message mode (SOCK\_RDM), the transport indicates when a partial message is received, either with the WSAEMSGSIZE error, or by setting the MSG\_PARTIAL flag upon return from the [WSARecv](#) and [WSARecvFrom](#) functions. When the last fragment of the full message is returned to the client, the error or flag is not indicated.

When the session is terminated gracefully, the receive operation fails with WSAEDISCON. When data loss occurs in the transport, PGM temporarily buffers the out-of-sequence packets and attempts to recover the lost data. If the data-loss is unrecoverable, the receive operation fail with WSAECONNRESET, and the session is terminated. The session can be reset due to a variety of conditions, including the following:

- The receiver or the incoming connection rate is too slow to keep pace with the incoming data rate.
- Excessive data loss occurs, possibly due to transient network conditions, such as routing problems, network instability, and so forth.
- An unrecoverable error occurs on the sender.
- Excessive resource utilization occurs on the local computer, such as exceeding the maximum allowed internal buffer storage, or encountering an out-of-resources condition.
- A data consistency check error occurs.
- Failure in a component PGM depends on, such as TCP/IP or Windows Sockets.

Both the first and second items in the above list could result in the receiver performing excessive buffering before running out of resources, or before ultimately moving beyond the sender's window.

## Terminating A PGM Session

The PGM sender or receiver can stop sending or receiving data by calling [closesocket](#). The receiver must call [closesocket](#) on both the listening and receiving sockets to prevent handle leaks. Calling [shutdown](#) on the sender before calling [closesocket](#) ensures all data gets sent, and ensures repair data is maintained until the send window advances past the last data sequence, even if the application itself terminates.

# Multihoming and PGM

3/5/2021 • 2 minutes to read • [Edit Online](#)

Special consideration must be given to multihomed PGM senders or receivers. This page describes the considerations, and provides guidelines for best programming practices.

## Multihomed PGM Sender

When an application fails to specify an interface upon calling the **connect** function, the first available interface is used. If no interface is available, **connect** fails.

When an application specifies an interface using the **RM\_SET\_SEND\_IF** socket option, a **bind** attempt is made implicitly to that interface using TCP/IP, and fails if TCP/IP fails the bind request. If the interface is set using **RM\_SET\_SEND\_IF** multiple times, only the last interface set successfully is applicable.

Windows Sockets maintains which interface is set, and if that interface disappears, the session is disconnected.

## Multihomed PGM Receiver

When an application fails to specify an interface upon calling the **listen** function, the default interface is used. If no interface is available, **bind** fails.

When an application specifies one or more interfaces on which to listen, using **RM\_ADD\_RECEIVE\_IF**, Windows Sockets attempts to bind to the requested interface or interfaces using TCP/IP. Any error from TCP/IP causes this request to fail. Unlike the PGM sender case, adding a receive interface multiple times result in the listens being posted on all the successfully added interfaces. Use the **RM\_DEL\_RECEIVE\_IF** socket option to stop listening on an interface.

Windows Sockets does not maintain state about multiple specified listening interfaces, and instead relies on TCP/IP to do so. Once a session is in progress, however, Windows Sockets track the incoming interface for that session, and if that interface disappears, Windows Sockets disconnects the session.

# PGM Socket Options

3/5/2021 • 2 minutes to read • [Edit Online](#)

PGM uses socket options to set state, provide multicast parameters, and otherwise implement its multicast capabilities. This page specifies how PGM socket options should be set, enumerates the socket options available for PGM, and where appropriate, provides usage examples and additional information for various options. For basic definitions of each PCM socket option, see [Socket Options](#).

**Windows XP:** Reliable Multicast Programming (PGM) is not supported.

The following socket options are available for PGM senders:

RM\\_LATEJOIN RM\\_RATE\\_WINDOW\\_SIZE RM\\_SEND\\_WINDOW\\_ADV\\_RATE RM\\_SENDER\\_STATISTICS  
RM\\_SENDER\\_WINDOW\\_ADVANCE\\_METHOD RM\\_SET\\_MCAST\\_TTL RM\\_SET\\_MESSAGE\\_BOUNDARY  
RM\\_SET\\_SEND\\_IF RM\\_USE\\_FEC

The RM\_SENDER\_WINDOW\_ADVANCE\_METHOD option specifies the method used when advancing the trailing edge send window. The optval parameter can only be E\_WINDOW\_ADVANCE\_BY\_TIME (the default). Note that E\_WINDOW\_USE\_AS\_DATA\_CACHE is not supported.

The following socket options are available for PGM receivers:

RM\\_ADD\\_RECEIVE\\_IF RM\\_DEL\\_RECEIVE\\_IF RM\\_HIGH\\_SPEED\\_INTRANET\\_OPT RM\\_RECEIVER\\_STATISTICS

## Setting PGM Socket Options

The following code snippet illustrates a programming guideline for setting PGM socket options:

```
ULONG      OptionData;    // This structure is option-dependent
//      :
setsockopt (s,
            IPPROTO_RM,
            Socket_Option,
            (char *) &OptionData,
            sizeof (OptionData));
```

In the snippet above, the type and contents of *OptionData* are dependent on the socket option being set. For all PGM socket options, the socket level is IPPROTO\_RM. PGM socket options must be set immediately following the call to the **bind** function, with the following exceptions:

RM\\_SET\\_MESSAGE\\_BOUNDARY RM\\_SENDER\\_STATISTICS RM\\_RECEIVER\\_STATISTICS

# Winsock Tracing

3/5/2021 • 2 minutes to read • [Edit Online](#)

## Introduction

Winsock tracing is a troubleshooting feature that can be enabled in retail binaries to trace certain Windows socket events with minimal overhead. The goal of adding retail tracing to Windows Sockets is to allow for better diagnostic capabilities for developers and product support. Winsock network event tracing supports tracing socket operations for IPv4 and IPv6 applications. Winsock catalog change tracing supports tracing changes made to the Winsock catalog by layered service providers (LSPs). Winsock tracing is supported on Windows Vista and later.

### NOTE

Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use [Windows Filtering Platform](#).

When an unexpected error occurs on a socket, the main clue to diagnose the problem is the error code returned. Very often, the returned error code does not explain why the error happened, especially when the error is initiated by the underlying network transport. Winsock tracing provides a more verbose tracing level which can log additional information to catch buffer corruption and poorly written applications.

Winsock tracing uses Event Tracing for Windows (ETW), a general-purpose, high-speed tracing facility provided by the operating system. Using a buffering and logging mechanism implemented in the kernel, ETW provides a tracing mechanism for events raised by both user-mode applications and kernel-mode device drivers. Additionally, ETW gives you the ability to enable and disable logging dynamically, making it easy to perform detailed tracing in production environments without requiring reboots or application restarts. The logging mechanism uses buffers that are written to disk by an asynchronous writer thread. This allows large-scale server applications to write events with minimum disturbance. ETW was first introduced on Windows 2000. Support for Winsock tracing using ETW was added on Windows Vista and later. For general information on ETW, see [Improve Debugging And Performance Tuning With ETW](#).

Winsock tracing can only be enabled at the operating system level for all processes and threads running on a computer. Winsock tracing currently cannot be enabled for just a single process or thread. When Winsock network event tracing is enabled, all socket applications (both IPv4 and IPv6) on a computer are traced.

The following topics describe Winsock tracing in more detail:

- [Winsock Tracing Levels](#)
- [Control of Winsock Tracing](#)
- [Winsock Network Event Tracing Details](#)
- [Winsock Catalog Change Tracing Details](#)

## Related topics

[Improve Debugging And Performance Tuning With ETW](#)

[Debug and Trace Facilities](#)



# Winsock Tracing Levels

3/5/2021 • 2 minutes to read • [Edit Online](#)

## Levels of Winsock Tracing

There are two levels of logging possible in Winsock tracing:

- Information
- Verbose

The information level traces socket create and close events, as well as any errors that occur on the socket.

The verbose level includes the information level events, and adds additional tracing for send and receive events. The verbose logging would be used to catch buffer corruption issues as well as poorly written applications.

Either the information or verbose level can be used with the Winsock Network Event tracing. The Winsock Catalog Change tracing supports only information level.

## Information Event Tracing

The following list details those Winsock network event socket operations that are traced at the information level:

- Socket creation

An event is logged on socket creation which can be used to trace the lifetime of a socket. These events also includes sockets created by accepting connections on a listening socket.

- Bind

The local IP address is logged to help correlate the Winsock tracing information to an application's socket calls.

- Connect

The remote IP address of the connected socket is logged to help correlate the Winsock tracing information to an application's socket calls.

- Winsock-initiated aborts and cancels

Anytime Winsock actively aborts or cancels a request, the event is logged.

- Transport initiated resets

Anytime the underlying transport indicates a connection has been reset, the event is logged.

- Send and receive errors

Whenever a send or receive call to the underlying transport fails, the event is logged.

- Socket disconnect and close

An event is logged when a socket handle is closed.

## Verbose Event Tracing

All of the information events are traced at the verbose level. The following list details those additional Winsock network event socket operations that are traced at the verbose level:

- Send and receive buffers

Events are logged of user buffer addresses and lengths when send and receive calls are posted to Winsock, as well as upon completion of these calls. This is useful for diagnosing buffer re-use issues as well as inefficient use of buffers.

- Socket options

An event is logged when an application changes certain socket option values. Some of the options logged include SO\_SNDBUF, SO\_RCVBUF, SIO\_ENABLE\_CIRCULAR\_QUEUEING, and FIONBIO.

- WSAPoll and select

An event is logged of an application's usage of **WSAPoll** and **select** calls which can be used to find performance bottlenecks.

- Winsock-initiated aborts and cancels

Anytime Winsock actively aborts or cancels a request, the event is logged.

- Event mask

An event is logged of the event mask an application registers for using the **WSAEventSelect** function.

- Datagram

An event is logged whenever a datagram arrives and there is no buffer space in which to copy it.

## Related topics

[Control of Winsock Tracing](#)

[Winsock Tracing](#)

[Winsock Catalog Change Tracing Details](#)

[Winsock Network Event Tracing Details](#)

# Control of Winsock Tracing

4/27/2021 • 6 minutes to read • [Edit Online](#)

Winsock tracing can be controlled by using either of the following methods:

- Command-line tools

Two command-line tools are included with Windows Vista and Windows Server 2008 that are used to control tracing and convert the binary trace log file to readable text.

The **logman.exe** tool is used to start or stop Winsock tracing.

The **tracert.exe** tool is used to convert the binary trace log file to a readable text file.

- Event Viewer

The Event Viewer on Windows Vista and later can also be used to enable Winsock tracing. The Event Viewer is accessible under the Administrative Tools from the Start menu.

## Using logman and tracert

Winsock network event tracing is disabled by default on Windows Vista and later.

The following command starts Winsock network event tracing on a computer, sets the name of event trace session to mywinsocksession, and sends output to a binary log file called winsocklogfile.etl:

**logman start -ets mywinsocksession -o winsocklogfile.etl -p Microsoft-Windows-Winsock-AFD**

Log files are created in the current directory with filenames of the form winsocklogfile\_000001.etl

The following command stops the above Winsock tracing on a computer for the session named mywinsocksession:

**logman stop -ets mywinsocksession**

A binary log file will be written to the location specified by the **-o** parameter. To translate the binary file into a readable text file, **tracert.exe** is used:

**tracert.exe <name of the .etl file> -o winsocktracelog.txt**

If an output file containing xml rather than plain text is preferred, the following command is used:

**tracert.exe <name of the .etl file> -o winsocktracelog.xml -of xml**

Winsock catalog change tracing is enabled by default on Windows Vista and later.

### NOTE

Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use [Windows Filtering Platform](#).

The following command starts Winsock Catalog Change tracing for layered service providers (LSPs) on a computer, sets the name of event trace session to mywinsockcatalogsession, and sends output to a binary log file called winsockcataloglogfile.etl:

```
logman start -ets mywinsockcatalogsession -o winsockcataloglogfile.etl -p Microsoft-Windows-Winsock-WS2HELP
```

Log files are created in the current directory with filenames of the form winsockcataloglogfile\_000001.etl

The following command stops the above Winsock tracing on a computer for the session named mysession:

```
logman stop -ets mywinsockcatalogsession
```

A binary log file will be written to the location specified by the **-o** parameter. To translate the binary file into a readable text file, **tracerpt.exe** is used:

```
tracerpt.exe <name of the .etl file> -o winsockcatalogtracelog.txt
```

If an output file containing xml rather than plain text is preferred, the following command is used:

```
tracerpt.exe <name of the .etl file> -o winsockcatalogtracelog.xml -of xml
```

## Using Event Viewer to Start Winsock Network Event Tracing

When you open Event Viewer, the left pane contains the list of events. Open **Applications and Services Logs** and navigate to **Microsoft\Windows\Winsock Network Event** as the source and select **Operational**.

In the Action pane, select **Log Properties** and check the **Enable Logging** check box. Once logging is enabled, you can also change the size of the log file if this is needed.

Winsock network event tracing is now enabled and all you need to do is hit the Refresh action to update the list of events that have been logged. To stop logging, simply uncheck the same radio button.

You may need to increase the log size depending on how many events you want to see. One drawback to using the Event Viewer for Winsock tracing is that it does not load all the string resources so the messages displayed in the Description field (once you select an event) is sometimes hard to read (an argument that should be formatted as hex will be displayed in decimal, for example). However, you can select the **Details** tab in the event description which shows the raw XML log entry which usually has easier to understand arguments.

## Using Event Viewer to Start Winsock Catalog Change Tracing

When you open Event Viewer, the left pane contains the list of events. Open **Applications and Services Logs** and navigate to **Microsoft\Windows\Winsock Catalog Change** as the source and select **Operational**.

In the Action pane, select **Log Properties** and check the **Enable Logging** check box. Once logging is enabled, you can also change the size of the log file if this is needed.

Winsock catalog change tracing is now enabled and all you need to do is hit the Refresh action to update the list of events that have been logged. To stop logging, simply uncheck the same radio button.

You may need to increase the log size depending on how many events you want to see. One drawback to using the Event Viewer for Winsock tracing is that it does not load all the string resources so the messages displayed in the Description field (once you select an event) is sometimes hard to read (an argument that should be formatted as hex will be displayed in decimal, for example). However, you can select the **Details** tab in the event description which shows the raw XML log entry which usually has easier to understand arguments.

## Interpreting Winsock Tracing Logs

All Winsock tracing events in a log contain two types of information:

- System
- EventData

The system information contains the logging level, the time the log entry was created, the event ID that represents the event type, the execution Process ID, the execution Thread ID, and other system information. A log level of 4 in Winsock tracing represents information event logging. A log level of 5 in Winsock tracing represents verbose event logging.

The execution process ID and thread ID in the system information indicates the process and thread that was running when the event occurred. In many cases, this will represent a kernel or worker thread and process, not a user-mode thread and or the process of the application. So this field is normally not very useful.

Each Winsock tracing event type has a unique event ID in the system section of the logged data. These event IDs can easily be used to filter a log file for specific Winsock tracing events.

The eventdata contains information specific to the event type.

The Process parameter in the eventdata information is the kernel EPROCESS structure address for the process, not the actual PID. To match an event to the user mode PID, take the Process value from the eventdata information from any log entry and look earlier in the log for a socket creation event with the Process value. Once a match is found, the last parameter in the socket creation event data is the user-mode Process ID that created the socket.

An Address parameter in the eventdata information is returned in some Winsock tracing events. An Address parameter represents an IP address, but it is displayed in the text file created by the **tracerpt.exe** tool or in Event Viewer as raw bytes or a number. IPv6 addresses are displayed in hexadecimal, so they are more easily understood. IPv4 addresses are displayed as a large decimal number. Developers will need to manually convert the raw bytes of an IPv4 address to the more familiar IPv4 dotted-decimal address notation to be better able to interpret the value.

An Error parameter in the eventdata is returned in some Winsock tracing events. An Error parameter is of the form of an NTSTATUS or HRESULT error code. This error parameter is displayed in the text file created by the **tracerpt.exe** tool or in Event Viewer as a decimal number. Developers will need to manually convert the decimal number to a hex number in order to better interpret the error code in some cases.

## Related topics

[Winsock Tracing](#)

[Winsock Catalog Change Tracing Details](#)

[Winsock Network Event Tracing Details](#)

[Winsock Tracing Levels](#)

# Winsock Network Event Tracing Details

3/5/2021 • 19 minutes to read • [Edit Online](#)

The following details each of the Winsock network events that can be traced and describes which parameters and information are logged.

## Socket Creation

Event ID = 1

Level = 4 (Information)

The following Winsock events are traced for socket creation:

- Socket handles created by calls to the [socket](#) or [WSASocket](#) functions.
- Accepted socket handles on listening sockets.
- Socket handles created by calls to the [WSAJoinLeaf](#) function.
- Socket handles re-used by calls to the [AcceptEx](#) or [ConnectEx](#) functions.

The following parameters are logged for a socket creation event:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
SocketType	The type of the socket.
Protocol	The protocol of the socket.
UserModePid	The user-mode process ID that created the socket.

## Socket Bind

Event ID = 2 (IPv4), Event ID = 3 (IPv6)

Level = 4 (Information)

The following Winsock events are traced for a bind operation:

- Implicit or explicit binding of a socket handle.

The following parameters are logged for a bind event:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.

PARAMETER	DESCRIPTION
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Address	The local IP address.
Port	The local IP port number.
Status	The status or error code returned for the bind operation.

## Failed Bind

Event ID = 40

Level = 4 (Information)

The following Winsock events are traced for a failed bind operation:

- Implicit or explicit binding of a socket handle that fails.

The following parameters are logged for a failed bind event:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Error	The error code returned for the failing bind operation.

## Socket Connect

Event ID = 4 (IPv4), Event ID = 5 (IPv6)

Level = 4 (Information)

The following Winsock events are traced for a connect operation request (a call to the [connect](#), [ConnectEx](#), [WSAConnect](#), [WSAConnectByList](#), or [WSAConnectByName](#) function):

- Connecting a socket to a destination for either a connection-oriented or a connectionless socket.

The following parameters are logged for a connect event:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.

PARAMETER	DESCRIPTION
Address	The remote IP address.
Port	The remote IP port number.

## Connect Completed

Event ID = 6

Level = 4 (Information)

The following Winsock events are traced for a connect completed:

- The connect operation is completed.

The following parameters are logged for a connect completed event:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Error	The error code returned for the connect operation.

## AFD-Initiated Abort

Event ID = 7

Level = 4 (Information)

The following Winsock events are traced for Winsock-initiated aborts or cancel operations:

- An abort due to unread receive data buffered after close.
- An abort after a call to the [shutdown](#) function with the *how* parameter set to SD\_RECEIVE and a call to the [closesocket](#) function with receive data pending.
- An abort after a failed attempt to flush the endpoint.
- An abort after an internal Winsock error occurred.
- An abort due to a connection with errors and the application previously requested that the connection be aborted on certain circumstances. One example of this case would be an application that set SO\_LINGER with a timeout of zero and there is still unacknowledged data on the connection.
- An abort on a connection not fully associated with accepting endpoint.
- An abort on a failed call to the [accept](#) or [AcceptEx](#) function.
- An abort due to a failed receive operation.
- An abort due to a Plug and Play event.
- An abort due to a failed flush request.
- An abort due to a failed expedited data receive request.
- An abort due to a failed send request.

- An abort due to canceled send request.
- An abort due to a canceled call to the [TransmitPackets](#) function.

The following parameters are logged for a Winsock-initiated abort or cancel operation:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Reason	The reason for the abort or cancel operation.

## Transport-Initiated Abort

Event ID = 8

Level = 4 (Information)

The following Winsock events are traced for transport-initiated abort or cancel operations:

- Reset indicated by the transport.

The following parameters are logged for a Winsock-initiated abort or cancel operation:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Reason	The reason for the abort or cancel operation.

## Failed Send Request

Event ID = 9

Level = 4 (Information)

The following Winsock events are traced for errors on [send](#) or [WSASend](#) requests:

- Errors returned on failed [send](#) or [WSASend](#) requests.

The following parameters are logged for a send requests that results in an error:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.

PARAMETER	DESCRIPTION
Error	The error code returned for the operation.

## Failed WsaSendMsg Request

Event ID = 10

Level = 4 (Information)

The following Winsock events are traced for errors on [WSASendMsg](#) requests:

- Errors returned on failed [WSASendMsg](#) requests.

The following parameters are logged for a send requests that results in an error:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Error	The error code returned for the operation.

## Failed Recv Request

Event ID = 11

Level = 4 (Information)

The following Winsock events are traced for errors on [recv](#), [WSARecv](#), or [WSARecvEx](#) requests:

- Errors returned on failed receive requests.

The following parameters are logged for a send requests that results in an error:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Error	The error code returned for the operation.

## Failed Recvfrom Request

Event ID = 12

Level = 4 (Information)

The following Winsock events are traced for errors on `recvfrom` or `WSARecvFrom` requests:

- Errors returned on failed `recvfrom` or `WSARecvFrom` requests.

The following parameters are logged for a `recvfrom` or `WSARecvFrom` request that results in an error:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Error	The error code returned for the operation.

## Socket Close

Event ID = 13

Level = 4 (Information)

The following Winsock events are traced for socket close operations:

- A socket handle is closed.

The following parameters are logged for a socket close event:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Error	The return value for the socket close operation.

## Socket Cleanup

Event ID = 14

Level = 4 (Information)

The following Winsock events are traced for socket cleanup (shutdown) operations:

- The `shutdown` function is called on a socket.
- The transport indicates a failed graceful disconnect.

The following parameters are logged for a socket cleanup (shutdown) or socket close event:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.

PARAMETER	DESCRIPTION
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Error	The return value for the socket cleanup (shutdown) operation.

## Socket Accept

Event ID = 15 (IPv4), Event ID = 16 (IPv6)

Level = 4 (Information)

The following Winsock events are traced for an [accept](#), [AcceptEx](#), or [WSAAccept](#) function request:

- An [accept](#), [AcceptEx](#), or [WSAAccept](#) function request on a socket handle.

The following parameters are logged for an accept event:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Address	The remote IP address.
Port	The remote IP port number.
Status	The status or error code returned for the accept operation.

## Accept Failed

Event ID = 17

Level = 4 (Information)

The following Winsock events are traced for a failed accept operation:

- An [accept](#), [AcceptEx](#), or [WSAAccept](#) request on a socket handle that fails.

The following parameters are logged for a failed accept event:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.

PARAMETER	DESCRIPTION
Error	The error code returned for the failing accept operation.

## Send Posted

Event ID = 18

Level = 5 (Verbose)

In order to diagnose user buffer corruption (for example, when an application re-uses the same buffer in another send or receive call while it's still in use), the data buffer is logged when posted to Winsock and upon completion by the underlying transport. The following Winsock events are traced for socket send and receive buffer post operations:

- An application posts a send.
- A send operation completes to Winsock.

The following parameters are logged for socket send operations:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
FastPath	A Boolean value that indicates if fast path I/O was used.
BufferCount	The buffer count.
Buffer	The virtual address of the buffer. For chained buffers, this parameter is the virtual address of the first buffer in the chain.
BufferLength	The length of the buffer. For chained buffers, this parameter is the total number of bytes in all of the buffers in the chain.

When FastPath is true, the usermode address of the first buffer in the array of buffers is logged in the Buffer parameter. When FastPath is false, the Winsock kernel buffer address is logged in the Buffer parameter.

## Receive Posted

Event ID = 19

Level = 5 (Verbose)

In order to diagnose user buffer corruption (for example, when an application re-uses the same buffer in another send or receive call while it's still in use), the data buffer is logged when posted to Winsock and upon completion by the underlying transport. The following Winsock events are traced for socket receive buffer post operations:

- An application posts a receive.

- A receive operation completes to Winsock.

The following parameters are logged for socket receive operations:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
FastPath	A Boolean value that indicates if fast path I/O was used.
BufferCount	The buffer count.
Buffer	The virtual address of the buffer. For chained buffers, this parameter is the virtual address of the first buffer in the chain.
BufferLength	The length of the buffer. For chained buffers, this parameter is the total number of bytes in all of the buffers in the chain.

When FastPath is true, the usermode address of the first buffer in the array of buffers is logged in the Buffer parameter. When FastPath is false, the Winsock kernel buffer address is logged in the Buffer parameter.

## RecvFrom Posted

Event ID = 20

Level = 5 (Verbose)

In order to diagnose user buffer corruption (for example, when an application re-uses the same buffer in another send or receive call while it's still in use), the data buffer is logged when posted to Winsock and upon completion by the underlying transport. The following Winsock events are traced for a [recvfrom](#) buffer post operation on a socket:

- An application posts a receive from operation.

The following parameters are logged for the recvfrom operation:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
FastPath	A Boolean value that indicates if fast path I/O was used.
BufferCount	The buffer count.
Buffer	The virtual address of the buffer. For chained buffers, this parameter is the virtual address of the first buffer in the chain.

PARAMETER	DESCRIPTION
BufferLength	The length of the buffer. For chained buffers, this parameter is the total number of bytes in all of the buffers in the chain.

When FastPath is true, the usermode address of the first buffer in the array of buffers is logged in the Buffer parameter. When FastPath is false, the Winsock kernel buffer address is logged in the Buffer parameter.

## SendTo Posted

Event ID = 21 (IPv4), Event ID = 22 (IPv6)

Level = 5 (Verbose)

In order to diagnose user buffer corruption (for example, when an application re-uses the same buffer in another send or receive call while it's still in use), the data buffer is logged when posted to Winsock and upon completion by the underlying transport. The following Winsock events are traced for a [sendto](#) buffer post operation on a socket:

- An application posts a send from.

The following parameters are logged for the [sendto](#) operation:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
FastPath	A Boolean value that indicates if fast path I/O was used.
BufferCount	The buffer count.
Buffer	The virtual address of the buffer. For chained buffers, this parameter is the virtual address of the first buffer in the chain.
BufferLength	The length of the buffer. For chained buffers, this parameter is the total number of bytes in all of the buffers in the chain.
Address	The remote IP address of the socket.
Port	The remote IP port number of the socket.

When FastPath is true, the usermode address of the first buffer in the array of buffers is logged in the Buffer parameter. When FastPath is false, the Winsock kernel buffer address is logged in the Buffer parameter.

## Recv Completed

Event ID = 23

Level = 5 (Verbose)

In order to diagnose user buffer corruption (for example, when an application re-uses the same buffer in another send or receive call while it's still in use), the data buffer is logged when posted to Winsock and upon completion by the underlying transport. The following Winsock events are traced for socket receive completed operations:

- A send operation completes to the transport.
- A receive operation completes to the transport.

The following parameters are logged for a send completed or receive completed:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Buffer	The virtual address of the buffer. For chained buffers, this parameter is the virtual address of the first buffer in the chain.
BufferLength	The length of the buffer of bytes received. For chained buffers, this parameter is the total bytes received in all buffers in the chain.

## Send Completed

Event ID = 24

Level = 5 (Verbose)

In order to diagnose user buffer corruption (for example, when an application re-uses the same buffer in another send or receive call while it's still in use), the data buffer is logged when posted to Winsock and upon completion by the underlying transport. The following Winsock events are traced for socket send completed operations:

- A send operation completes to the transport.

The following parameters are logged for a send completed:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Buffer	The virtual address of the buffer. For chained buffers, this parameter is the virtual address of the first buffer in the chain.
BufferLength	The length of the buffer of bytes sent. For chained buffers, this parameter is the total bytes sent from all buffers in the chain.

## SendMsg Completed

Event ID = 25

Level = 5 (Verbose)

In order to diagnose user buffer corruption (for example, when an application re-uses the same buffer in another send or receive call while it's still in use), the data buffer is logged when posted to Winsock and upon completion by the underlying transport. The following Winsock events are traced when a [WSASendMsg](#) buffer post operation completes on a socket:

- An application completes a [WSASendMsg](#) operation.

The following parameters are logged for the [WSASendMsg](#) completion:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
BufferCount	The buffer count.
Buffer	The virtual address of the buffer. For chained buffers, this parameter is the virtual address of the first buffer in the chain.
BufferLength	The length of the buffer of bytes sent. For chained buffers, this parameter is the total bytes sent from all buffers in the chain.
Address	The remote IP address of the socket.
Port	The remote IP port number of the socket.

## RecvFrom Completed

Event ID = 26 (IPv4), Event ID = 27 (IPv6)

Level = 5 (Verbose)

In order to diagnose user buffer corruption (for example, when an application re-uses the same buffer in another send or receive call while it's still in use), the data buffer is logged when posted to Winsock and upon completion by the underlying transport. The following Winsock events are traced when a [recvfrom](#) buffer post operation completes on a socket:

- An application completes a [recvfrom](#) operation.

The following parameters are logged for the [recvfrom](#) completion:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
BufferCount	The buffer count.
Buffer	The virtual address of the buffer. For chained buffers, this parameter is the virtual address of the first buffer in the chain.
BufferLength	The length of the buffer of bytes received. For chained buffers, this parameter is the total bytes received in all buffers in the chain.
Address	The remote IP address of the socket.
Port	The remote IP port number of the socket.

## SendTo Completed

Event ID = 28

Level = 5 (Verbose)

In order to diagnose user buffer corruption (for example, when an application re-uses the same buffer in another send or receive call while it's still in use), the data buffer is logged when posted to Winsock and upon completion by the underlying transport. The following Winsock events are traced when a **sendto** buffer post operation completes on a socket:

- An application completes a **sendto** operation.

The following parameters are logged for the **sendto** completion:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
BufferCount	The buffer count.
Buffer	The virtual address of the buffer. For chained buffers, this parameter is the virtual address of the first buffer in the chain.
BufferLength	The length of the buffer of bytes sent. For chained buffers, this parameter is the total bytes sent from all buffers in the chain.

PARAMETER	DESCRIPTION
Address	The remote IP address of the socket.
Port	The remote IP port number of the socket.

## Socket Option Set

Event ID = 29

Level = 5 (Verbose)

Whenever an application changes certain socket option values and loctls, the new values will be logged. The options logged can be used to diagnose poor performance or strange behavior in applications. The following Winsock events are traced for certain socket options and loctls:

- SO\_SNDBUF changes.
- SO\_RCVBUF changes.
- FIONBIO
- SIO\_ENABLE\_CIRCULAR\_QUEUEING
- SIO\_UDP\_CONNRESET
- SO\_OOBINLINE

The following parameters are logged for [setsockopt](#) and [WSAloctl](#) function calls that change any of the above values:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Option	The socket option or loctl that is changed.
Value	The new value for the socket option or loctl.

## Select/Poll Posted

Event ID = 30

Level = 5 (Verbose)

The following Winsock events are traced when an application calls the [select](#) or [WSAPoll](#) function:

- Application posts a [select](#) or [WSAPoll](#) request.

The following parameters are logged for [select](#) or [WSAPoll](#) events:

PARAMETER	DESCRIPTION
Process	The owning process ID.
HandleCount	The number of handles passed in by the application (only valid on the initiating event).
Timeout	The maximum time for the <a href="#">select</a> or <a href="#">WSAPoll</a> function to wait.

## Select/Poll Completed

Event ID = 31

Level = 5 (Verbose)

The following Winsock events are traced when an application calls the [select](#) or [WSAPoll](#) function:

- Winsock completes a [select](#) or [WSAPoll](#) call.

The following parameters are logged when a [select](#) or [WSAPoll](#) operation completes:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Error	The error code returned for the <a href="#">select</a> or <a href="#">WSAPoll</a> operation.

## WSAEventSelect

Event ID = 32

Level = 5 (Verbose)

The following Winsock events are traced when an application calls the [WSAEventSelect](#) function:

- Log the event mask passed in the [WSAEventSelect](#) function.

The following parameters are logged for [WSAEventSelect](#) function calls:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
EventMask	The value for the event mask.

## Dropped Datagram

Event ID = 33 (IPv4), Event ID = 34 (IPv6)

Level = 5 (Verbose)

To help diagnose issues around datagram applications, the following Winsock events are traced:

- When a datagram arrives and it is dropped due to insufficient buffer space.
- On a connected datagram, if data arrives from a source other than connected destination.

The following parameters are logged for dropped datagrams:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
PacketSize	The size of the packet that was dropped.
Address	The IP address of the source of the packet.
Port	The IP port number of the source of the packet.
Reason	The error code or reason the packet was dropped.

## Connection Indicated

Event ID = 35 (IPv4), Event ID = 36 (IPv6)

Level = 5 (Verbose)

The following Winsock events are traced for connection indicated operations:

- An application receives a connection request.

The following parameters are logged for connections indicated from transport events:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Address	The remote IP address.
Port	The remote IP port number.

## Data Indicated

Event ID = 37

Level = 5 (Verbose)

The following Winsock events are traced for data indicated operations:

- An application receives data on a connected socket.

The following parameters are logged for data indicated from transport events:

PARAMETER	DESCRIPTION
Process	The owning process ID.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Bytes Indicated	The number of bytes received on the socket.

## Data Indicated from Transport

Event ID = 38 (IPv4), Event ID = 39 (IPv6)

Level = 5 (Verbose)

The following Winsock events are traced for data indicated from transport operations:

- An application posts a receive request and receives data.

The following parameters are logged for data indicated from transport events:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.
Address	The remote IP address.
Port	The remote IP port number.
Bytes Indicated	The number of bytes received on the socket.

## Disconnect Indicated from Transport

Event ID = 41

Level = 5 (Verbose)

The following Winsock events are traced for disconnect indicated operations:

- An application receives a disconnect indication.

The following parameters are logged for disconnect indicated from transport events:

PARAMETER	DESCRIPTION
Process	The kernel EPROCESS structure address for the process.
Endpoint	The Winsock kernel socket address used as a unique identifier for a socket.

## Related topics

[Control of Winsock Tracing](#)

[Winsock Tracing](#)

[Winsock Catalog Change Tracing Details](#)

[Winsock Tracing Levels](#)

# Winsock Catalog Change Tracing Details

3/5/2021 • 2 minutes to read • [Edit Online](#)

## NOTE

Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use [Windows Filtering Platform](#).

Winsock Catalog Change event tracing for layered Service providers (LSPs) is related to LSP installation, LSP removal, LSP disable, and Winsock catalog reset operations. All of the following events are written to the *Microsoft-Windows-Winsock-WS2HELP/Operational* channel which is different from the other Winsock network event tracing logged on Windows Vista and later.

The following details each of the Winsock LSP events that can be traced and describes which parameters and information are logged.

## LSP Install

Event ID = 1

Level = 4 (Information)

The following Winsock LSP events are traced for an LSP install operation:

- A protocol entry is installed into the Winsock catalog.

The following parameters are logged for a LSP install event:

PARAMETER	DESCRIPTION
LSP Name	The name of the LSP as obtained from the <code>szProtocol</code> member of the <a href="#">WSAPROTOCOL_INFO</a> structure for the LSP being installed.
Catalog	The Winsock catalog (32-bit or 64-bit) where the LSP is being installed. This is an integer value that is either 32 or 64.
Installer	The module filename of the application making the LSP install call.
GUID	The GUID value of the Winsock transport provider that the LSP is being installed under.
Category	The <code>dwCatalogEntryId</code> member of the <a href="#">WSAPROTOCOL_INFO</a> structure for the LSP being installed.

## LSP Uninstall

Event ID = 2

Level = 4 (Information)

The following Winsock LSP events are traced for an LSP uninstall operation:

- A protocol entry is removed from the Winsock catalog.

The following parameters are logged for a LSP uninstall event:

PARAMETER	DESCRIPTION
LSP Name	The name of the LSP as obtained from the <code>szProtocol</code> member of the <a href="#">WSAPROTOCOL_INFO</a> structure for the LSP being removed.
Catalog	The Winsock catalog (32-bit or 64-bit) where the LSP is being removed. This is an integer value that is either 32 or 64.
Installer	The module filename of the application making the LSP remove call.
GUID	The GUID value of the Winsock transport provider that the LSP is removed from.
Category	The <code>dwCatalogEntryId</code> member of the <a href="#">WSAPROTOCOL_INFO</a> structure for the LSP being removed.

## LSP Disable

Event ID = 3

Level = 4 (Information)

The following Winsock LSP events are traced for an LSP disable operation:

- A protocol entry is disabled in the Winsock catalog.

The following parameters are logged for a LSP disable event:

PARAMETER	DESCRIPTION
LSP Name	The name of the LSP as obtained from the <code>szProtocol</code> member of the <a href="#">WSAPROTOCOL_INFO</a> structure for the LSP being disabled.
Catalog	The Winsock catalog (32-bit or 64-bit) where the LSP is being disabled. This is an integer value that is either 32 or 64.
Installer	The module filename of the application making the LSP disable call.

PARAMETER	DESCRIPTION
GUID	The GUID value of the Winsock transport provider where the LSP is being disabled.
Category	The <b>dwCatalogEntryId</b> member of the <a href="#">WSAPROTOCOL_INFO</a> structure for the LSP being disabled.

## Winsock Catalog Reset

Event ID = 4

Level = 4 (Information)

The following Winsock LSP events are traced for a Winsock catalog reset operation:

- The Winsock catalog is reset.

The following parameters are logged for a Winsock catalog reset event:

PARAMETER	DESCRIPTION
Catalog	The Winsock catalog (32-bit or 64-bit) that is being reset. This is an integer value that is either 32 or 64.

## Related topics

[Control of Winsock Tracing](#)

[Winsock Tracing](#)

[Winsock Tracing Levels](#)

[Winsock Network Event Tracing Details](#)

# Winsock timestamping

5/26/2021 • 6 minutes to read • [Edit Online](#)

## Introduction

Packet timestamps are a crucial feature to many clock synchronization applications—for example, Precision Time Protocol. The closer the timestamp generation is to when a packet is received/sent by the network adapter hardware, the more accurate the synchronization application can be.

So the timestamping APIs described in this topic provide your application with a mechanism to report timestamps that are generated well below the application layer. Specifically, a software timestamp at the interface between the miniport and NDIS, and a hardware timestamp in the NIC hardware. The timestamping API can greatly improve clock synchronization accuracy. Currently, support is scoped to User Datagram Protocol (UDP) sockets.

## Receive timestamps

You configure receive timestamp reception through the [SIO\\_TIMESTAMPING](#) IOCTL. Use that IOCTL to enable receive timestamp reception. When you receive a datagram using the [LPFN\\_WSARECVMSG \(WSARecvMsg\)](#) function, its timestamp (if available) is contained in the SO\_TIMESTAMP control message.

SO\_TIMESTAMP (0x300A) is defined in [mstcpip.h](#). The control message data is returned as a [UINT64](#).

## Transmit timestamps

Transmit timestamp reception is also configured through the [SIO\\_TIMESTAMPING](#) IOCTL. Use that IOCTL to enable transmit timestamp reception, and specify the number of transmit timestamps that the system will buffer. As transmit timestamps are generated, they are added to the buffer. If the buffer is full, new transmit timestamps are discarded.

When sending a datagram, associate the datagram with an SO\_TIMESTAMP\_ID control message. This should contain a unique identifier. Send the datagram, along with its SO\_TIMESTAMP\_ID control message, using [WSASendMsg](#). Transmit timestamps might not be immediately available after [WSASendMsg](#) returns. As transmit timestamps become available, they are placed into a per-socket buffer. Use the [SIO\\_GET\\_TX\\_TIMESTAMP](#) IOCTL to poll for the timestamp by its ID. If the timestamp is available, then it is removed from the buffer and returned. If the timestamp is not available, then [WSAGetLastError](#) returns [WSAEWOULDBLOCK](#). If a transmit timestamp is generated while the buffer is full, the new timestamp is discarded.

SO\_TIMESTAMP\_ID (0x300B) is defined in [mstcpip.h](#). You should supply the control message data as a [UINT32](#).

Timestamps are represented as a 64-bit counter value. The frequency of the counter depends on the source of the timestamp. For software timestamps, the counter is a [QueryPerformanceCounter](#) (QPC) value, and you can determine its frequency via [QueryPerformanceFrequency](#). For NIC hardware timestamps, the counter frequency is dependent on the NIC hardware, and you can determine it with additional information given by [CaptureInterfaceHardwareCrossTimestamp](#). To determine the source of timestamps, use the [GetInterfaceActiveTimestampCapabilities](#) and [GetInterfaceSupportedTimestampCapabilities](#) functions.

In addition to socket-level configuration using the [SIO\\_TIMESTAMPING](#) socket option to enable timestamp reception for a socket, system-level configuration is also needed.

## Estimating latency of socket send path

In this section, we'll use transmit timestamps to estimate the latency of the socket send path. If you have an existing application that consumes application-level IO timestamps—where the timestamp needs to be as close as possible to the actual point of transmission—then this sample provides a quantitative description as to how much the Winsock timestamping APIs can improve the accuracy of your application.

The example assumes that there's only one network interface card (NIC) in the system, and that *interfaceLuid* is the LUID of that adapter.

```
void QueryHardwareClockFrequency(LARGE_INTEGER* clockFrequency)
{
    // Returns the hardware clock frequency. This can be calculated by
    // collecting crosstimestamps via CaptureInterfaceHardwareCrossTimestamp
    // and forming a linear regression model.
}

void estimate_send_latency(SOCKET sock,
    PSOCKADDR_STORAGE addr,
    NET_LUID* interfaceLuid,
    BOOLEAN hardwareTimestampSource)
{
    DWORD numBytes;
    INT error;
    CHAR data[512];
    CHAR control[WSA_CMSG_SPACE(sizeof(UINT32))] = { 0 };
    WSABUF dataBuf;
    WSABUF controlBuf;
    WSAMSG wsaMsg;
    ULONG64 appLevelTimestamp;

    dataBuf.buf = data;
    dataBuf.len = sizeof(data);
    controlBuf.buf = control;
    controlBuf.len = sizeof(control);
    wsaMsg.name = (PSOCKADDR)addr;
    wsaMsg.namelen = (INT)INET_SOCKADDR_LENGTH(addr->ss_family);
    wsaMsg.lpBuffers = &dataBuf;
    wsaMsg.dwBufferCount = 1;
    wsaMsg.Control = controlBuf;
    wsaMsg.dwFlags = 0;

    // Configure tx timestamp reception.
    TIMESTAMPING_CONFIG config = { 0 };
    config.flags |= TIMESTAMPING_FLAG_TX;
    config.txTimestampsBuffered = 1;
    error =
        WSAIoctl(
            sock,
            SIO_TIMESTAMPING,
            &config,
            sizeof(config),
            NULL,
            0,
            &numBytes,
            NULL,
            NULL);
    if (error == SOCKET_ERROR) {
        printf("WSAIoctl failed %d\n", WSAGetLastError());
        return;
    }

    // Assign a tx timestamp ID to this datagram.
    UINT32 txTimestampId = 123;
    PCMSGHDR cmsp = WSA_CMSG_FTRSHDR(&wsaMsg);
```

```

    /* Create CMSG */
    cmsg->cmsg_len = WSA_CMSG_LEN(sizeof(UINT32));
    cmsg->cmsg_level = SOL_SOCKET;
    cmsg->cmsg_type = SO_TIMESTAMP_ID;
    *(PUINT32)WSA_CMSG_DATA(cmsg) = txTimestampId;

    // Capture app-layer timestamp prior to send call.
    if (hardwareTimestampSource) {
        INTERFACE_HARDWARE_CROSSTIMESTAMP crossTimestamp = { 0 };
        crossTimestamp.Version = INTERFACE_HARDWARE_CROSSTIMESTAMP_VERSION_1;
        error = CaptureInterfaceHardwareCrossTimestamp(interfaceLuid, &crossTimestamp);
        if (error != NO_ERROR) {
            printf("CaptureInterfaceHardwareCrossTimestamp failed %d\n", error);
            return;
        }
        appLevelTimestamp = crossTimestamp.HardwareClockTimestamp;
    }
    else { // software source
        LARGE_INTEGER t1;
        QueryPerformanceCounter(&t1);
        appLevelTimestamp = t1.QuadPart;
    }

    error =
        sendmsg(
            sock,
            &wsaMsg,
            0,
            &numBytes,
            NULL,
            NULL);
    if (error == SOCKET_ERROR) {
        printf("sendmsg failed %d\n", WSAGetLastError());
        return;
    }

    printf("sent packet\n");

    // Poll for the socket tx timestamp value. The timestamp may not be available
    // immediately.
    UINT64 socketTimestamp;
    ULONG maxTimestampPollAttempts = 6;
    ULONG txTimestampRetrieveIntervalMs = 1;
    BOOLEAN retrievedTimestamp = FALSE;
    for (ULONG i = 0; i < maxTimestampPollAttempts; i++) {
        error =
            WSAIoctl(
                sock,
                SIO_GET_TX_TIMESTAMP,
                &txTimestampId,
                sizeof(txTimestampId),
                &socketTimestamp,
                sizeof(socketTimestamp),
                &numBytes,
                NULL,
                NULL);
        if (error != SOCKET_ERROR) {
            ASSERT(numBytes == sizeof(timestamp));
            ASSERT(timestamp != 0);
            retrievedTimestamp = TRUE;
            break;
        }
    }

    error = WSAGetLastError();
    if (error != WSAEWOULDBLOCK) {
        printf("WSAIoctl failed % d\n", error);
        break;
    }
}

```

```

Sleep(1000); // Sleep for 1 second to let the system settle.

txTstampRetrieveIntervalMs *= 2;

}

if (retrievedTimestamp) {
    LARGE_INTEGER clockFrequency;
    ULONG64 elapsedMicroseconds;

    if (hardwareTimestampSource) {
        QueryHardwareClockFrequency(&clockFrequency);
    }
    else { // software source
        QueryPerformanceFrequency(&clockFrequency);
    }

    // Compute socket send path latency.
    elapsedMicroseconds = socketTimestamp - appLevelTimestamp;
    elapsedMicroseconds *= 1000000;
    elapsedMicroseconds /= clockFrequency.QuadPart;
    printf("socket send path latency estimation: %lld microseconds\n",
           elapsedMicroseconds);
}
else {
    printf("failed to retrieve TX timestamp\n");
}
}

```

## Estimating latency of socket receive path

Here's a similar sample for the receive path. The example assumes that there's only one network interface card (NIC) in the system, and that *interfaceLuid* is the LUID of that adapter.

```

void QueryHardwareClockFrequency(LARGE_INTEGER* clockFrequency)
{
    // Returns the hardware clock frequency. This can be calculated by
    // collecting crosstimestamps via CaptureInterfaceHardwareCrossTimestamp
    // and forming a linear regression model.
}

void estimate_receive_latency(SOCKET sock,
    NET_LUID* interfaceLuid,
    BOOLEAN hardwareTimestampSource)
{
    DWORD numBytes;
    INT error;
    CHAR data[512];
    CHAR control[WSA_CMSG_SPACE(sizeof(UINT64))] = { 0 };
    WSABUF dataBuf;
    WSABUF controlBuf;
    WSAMSG wsaMsg;
    UINT64 socketTimestamp = 0;
    ULONG64 appLevelTimestamp;

    dataBuf.buf = data;
    dataBuf.len = sizeof(data);
    controlBuf.buf = control;
    controlBuf.len = sizeof(control);
    wsaMsg.name = NULL;
    wsaMsg.namelen = 0;
    wsaMsg.lpbuffers = &dataBuf;
    wsaMsg.dwbufferCount = 1;
    wsaMsg.Control = controlBuf;
    wsaMsg.dwFlags = 0;

    // Configure rx timestamp reception.
    TIMESTAMPTING_CONFIG config = { 0 };

```

```

config.flags |= TIMESTAMPING_FLAG_RX;
error =
    WSAIoctl(
        sock,
        SIO_TIMESTAMPING,
        &config,
        sizeof(config),
        NULL,
        0,
        &numBytes,
        NULL,
        NULL);
if (error == SOCKET_ERROR) {
    printf("WSAIoctl failed %d\n", WSAGetLastError());
    return;
}

error =
    recvmsg(
        sock,
        &wsamsg,
        &numBytes,
        NULL,
        NULL);
if (error == SOCKET_ERROR) {
    printf("recvmsg failed %d\n", WSAGetLastError());
    return;
}

// Capture app-layer timestamp upon message reception.
if (hardwareTimestampSource) {
    INTERFACE_HARDWARE_CROSSTIMESTAMP crossTimestamp = { 0 };
    crossTimestamp.Version = INTERFACE_HARDWARE_CROSSTIMESTAMP_VERSION_1;
    error = CaptureInterfaceHardwareCrossTimestamp(interfaceLuid, &crossTimestamp);
    if (error != NO_ERROR) {
        printf("CaptureInterfaceHardwareCrossTimestamp failed %d\n", error);
        return;
    }
    appLevelTimestamp = crossTimestamp.HardwareClockTimestamp;
}
else { // software source
    LARGE_INTEGER t1;
    QueryPerformanceCounter(&t1);
    appLevelTimestamp = t1.QuadPart;
}

printf("received packet\n");

// Look for socket rx timestamp returned via control message.
BOOLEAN retrievedTimestamp = FALSE;
PCMSGHDR cmsg = WSA_CMSG_FIRSTHDR(&wsamsg);
while (cmsg != NULL) {
    if (cmsg->cmsg_level == SOL_SOCKET && cmsg->cmsg_type == SO_TIMESTAMP) {
        socketTimestamp = *(PUINT64)WSA_CMSG_DATA(cmsg);
        retrievedTimestamp = TRUE;
        break;
    }
    cmsg = WSA_CMSG_NXTHDR(&wsamsg, cmsg);
}

if (retrievedTimestamp) {
    // Compute socket receive path latency.
    LARGE_INTEGER clockFrequency;
    ULONG64 elapsedMicroseconds;

    if (hardwareTimestampSource) {
        QueryHardwareClockFrequency(&clockFrequency);
    }
    else { // software source

```

```
        QueryPerformanceFrequency(&clockFrequency);
    }

    // Compute socket send path latency.
    elapsedMicroseconds = appLevelTimestamp - socketTimestamp;
    elapsedMicroseconds *= 1000000;
    elapsedMicroseconds /= clockFrequency.QuadPart;
    printf("RX latency estimation: %lld microseconds\n",
        elapsedMicroseconds);
}
else {
    printf("failed to retrieve RX timestamp\n");
}
}
```

## A limitation

One limitation of the Winsock timestamping APIs is that calling [SIO\\_GET\\_TX\\_TIMESTAMP](#) is always a non-blocking operation. Even calling the IOCTL in an OVERLAPPED fashion results in an immediate return of [WSAEWOULDBLOCK](#) if there are currently no available transmit timestamps. Since transmit timestamps might not be immediately available after [WSASendMsg](#) returns, your application must poll the IOCTL until the timestamp is available. A transmit timestamp is guaranteed to be available after a successful [WSASendMsg](#) call given that the transmit timestamp buffer is not full.

# Winsock explicit congestion notification (ECN)

5/26/2021 • 3 minutes to read • [Edit Online](#)

## Introduction

Some applications and/or protocols that are based on the User Datagram Protocol (UDP) (for example, QUIC) seek to leverage the use of explicit congestion notification (ECN) codepoints in order to improve latency and jitter in congested networks.

The Winsock ECN APIs extend the `getsockopt/setsockopt` interface—as well as the [`WSASendMsg/LPFN\_WSARECVMMSG \(WSARecvMsg\)`](#) control message interface—with support for modifying and receiving ECN codepoints in IP headers. The functionality provided allows you to get and set ECN codepoints on a per-packet basis.

For more information regarding ECN, see [The Addition of Explicit Congestion Notification \(ECN\) to IP](#).

Your application isn't allowed to specify the Congestion Encountered (CE) code point when sending datagrams. The send will return with error **WSAEINVAL**.

## Query ECN with WSAGetRecvIPechn

[`WSAGetRecvIPechn`](#) is an inline function, defined in [`ws2tcpip.h`](#).

Call `WSAGetRecvIPechn` to query the current enablement of receiving the **IP\_ECN** (or **IPV6\_ECN**) control message via [`LPFN\_WSARECVMMSG \(WSARecvMsg\)`](#).

Also see the [\*\*WSAMSG\*\*](#) structure.

- **Protocol:** IPv4
- **Cmsg\_level:** IPPROTO\_IP
- **Cmsg\_type:** IP\_ECN (50 decimal)
- **Description:** Specifies/receives ECN codepoint in the Type of Service (TOS) IPv4 header field.
- **Protocol:** IPv6
- **Cmsg\_level:** IPPROTO\_IPV6
- **Cmsg\_type:** IPV6\_ECN (50 decimal)
- **Description:** Specifies/receives ECN codepoint in the Traffic Class IPv6 header field.

## Specify ECN with WSASetRecvIPechn

[`WSASetRecvIPechn`](#) is an inline function, defined in [`ws2tcpip.h`](#).

Call `WSASetRecvIPechn` to specify whether the IP stack should populate the control buffer with a message containing the ECN codepoint of the Type of Service IPv4 header field (or Traffic Class IPv6 header field) on a received datagram. When set to `TRUE`, the [`LPFN\_WSARECVMMSG \(WSARecvMsg\)`](#) function returns optional control data containing the ECN codepoint of the received datagram. The returned control message type will be **IP\_ECN** (or **IPV6\_ECN**) with level **IPPROTO\_IP** (or **IPPROTO\_IPV6**). The control message data is returned as an INT. This option is valid only on datagram sockets (the socket type must be **SOCK\_DGRAM**).

## Code example 1—application advertising ECN support

```
#define ECN_ECT_0 2

void sendEcn(SOCKET sock, PSOCKADDR_STORAGE addr, LPFN_WSASENDMSG sendmsg, PCHAR data, INT datalen)
{
    DWORD numBytes;
    INT error;

    CHAR control[WSA_CMSG_SPACE(sizeof(INT))] = { 0 };
    WSABUF dataBuf;
    WSABUF controlBuf;
    WSAMSG wsaMsg;
    PCMSGHDR cmsg;

    dataBuf.buf = data;
    dataBuf.len = datalen;
    controlBuf.buf = control;
    controlBuf.len = sizeof(control);
    wsaMsg.name = (PSOCKADDR)addr;
    wsaMsg.namelen = (INT)INET_SOCKADDR_LENGTH(addr->ss_family);
    wsaMsg.lpbuffers = &dataBuf;
    wsaMsg.dwBufferCount = 1;
    wsaMsg.Control = controlBuf;
    wsaMsg.dwFlags = 0;

    cmsg = WSA_CMSG_FIRSTHDR(&wsaMsg);
    cmsg->cmsg_len = WSA_CMSG_LEN(sizeof(INT));
    cmsg->cmsg_level = (addr->ss_family == AF_INET) ? IPPROTO_IP : IPPROTO_IPV6;
    cmsg->cmsg_type = (addr->ss_family == AF_INET) ? IP_ECN : IPV6_ECN;
    *(PINT)WSA_CMSG_DATA(cmsg) = ECN_ECT_0;

    error =
        sendmsg(
            sock,
            &wsaMsg,
            0,
            &numBytes,
            NULL,
            NULL);
    if (error == SOCKET_ERROR) {
        printf("sendmsg failed %d\n", WSAGetLastError());
    }
}
```

## Code example 2—application detecting congestion

```
#define ECN_ECT_CE 3

int recvEcn(SOCKET sock, PSOCKADDR_STORAGE addr, LPFN_WSARECVMMSG recvmsg, PCHAR data, INT datalen, PBOOLEAN
congestionEncountered)
{
    DWORD numBytes;
    INT error;
    INT ecnVal;
    SOCKADDR_STORAGE remoteAddr = { 0 };

    CHAR control[WSA_CMSG_SPACE(sizeof(INT))] = { 0 };
    WSABUF dataBuf;
    WSABUF controlBuf;
    WSAMSG wsaMsg;
    PCMSGHDR cmsg;

    dataBuf.buf = data;
    dataBuf.len = datalen;
```

```

dataBuf.len = dataLen;
controlBuf.buf = control;
controlBuf.len = sizeof(control);
wsaMsg.name = (PSOCKADDR)&remoteAddr;
wsaMsg.namelen = sizeof(remoteAddr);
wsaMsg.lpbuffers = &dataBuf;
wsaMsg.dwBufferCount = 1;
wsaMsg.Control = controlBuf;
wsaMsg.dwFlags = 0;

*congestionEncountered = FALSE;

error =
recvmsg(
    sock,
    &wsaMsg,
    &numBytes,
    NULL,
    NULL);
if (error == SOCKET_ERROR) {
    printf("recvmsg failed %d\n", WSAGetLastError());
    return -1;
}

cmsg = WSA_CMSG_FIRSTHDR(&wsaMsg);
while (cmsg != NULL) {
    if ((cmsg->cmsg_level == IPPROTO_IP && cmsg->cmsg_type == IP_ECN) ||
        (cmsg->cmsg_level == IPPROTO_IPV6 && cmsg->cmsg_type == IPV6_ECN)) {
        ecnVal = *(PINT)WSA_CMSG_DATA(cmsg);
        if (ecnVal == ECN_ECT_CE) {
            *congestionEncountered = TRUE;
        }
        break;
    }
    cmsg = WSA_CMSG_NXTHDR(&wsaMsg, cmsg);
}

return numBytes;
}

void receiver(SOCKET sock, PSOCKADDR_STORAGE addr, LPFN_WSARECVMSG recvmsg)
{
    DWORD numBytes;
    INT error;
    DWORD enabled;
    CHAR data[512];
    BOOLEAN congestionEncountered;

    error = bind(sock, (PSOCKADDR)addr, sizeof(*addr));
    if (error == SOCKET_ERROR) {
        printf("bind failed %d\n", WSAGetLastError());
        return;
    }

    enabled = TRUE;
    error = WSASetRecvIPEcn(sock, enabled);
    if (error == SOCKET_ERROR) {
        printf(" WSASetRecvIPEcn failed %d\n", WSAGetLastError());
        return;
    }

    do {
        numBytes = recvEcn(sock, addr, recvmsg, data, sizeof(data), &congestionEncountered);
        if (congestionEncountered) {
            // Tell sender to slow down
        }
    } while (numBytes > 0);
}

```

## See also

- [WSAGetRecvIPCEcn](#)
- [WSASetRecvIPCEcn](#)

# Winsock socket state notifications

5/26/2021 • 18 minutes to read • [Edit Online](#)

## Introduction

The socket state notifications APIs in the table below provide you with a scalable and efficient way to obtain notifications about socket state changes (efficient in terms of both CPU and memory). This includes notifications about things such as non-blocking read, non-blocking write, error conditions, and other info.

API	DESCRIPTION
<a href="#">ProcessSocketNotifications</a> function	Associates a set of sockets with a completion port, and retrieves any notifications that are already pending on that port. Once associated, the completion port receives the socket state notifications that were specified.
<a href="#">SOCK_NOTIFY_REGISTRATION</a> structure	Represents info supplied to the <a href="#">ProcessSocketNotifications</a> function.
<a href="#">SocketNotificationRetrieveEvents</a> function	This inline helper function is provided as a convenience to retrieve the events mask from an <a href="#">OVERLAPPED_ENTRY</a> .

The workflow begins with you associating sockets with an I/O completion port ([ProcessSocketNotifications](#) and [SOCK\\_NOTIFY\\_REGISTRATION](#)). After that, the port delivers info about socket state changes using the usual I/O completion port query methods.

These APIs allow easy construction of platform-agnostic abstractions. As such, persistent and one-shot, and level- and edge-triggered flags are supported. For example, one-shot level-triggered registrations are the recommended pattern for multi-threaded servers.

## Recommendations

These APIs provide a scalable alternative to the [WSAPoll](#) and [select](#) APIs.

They're an alternative to [overlapped socket I/O](#) used with [I/O completion ports](#), and they avoid the need for permanent per-socket I/O buffers. But in a scenario where per-socket I/O buffers aren't an important consideration (the number of sockets is relatively low, or they're constantly used), overlapped socket I/O might have less overhead due to a smaller number of kernel transitions, as well as a simpler model.

A socket may be associated with only a single I/O completion port. A socket may be registered with an I/O completion port only once. To change completion keys, deregister the notification, wait for the [SOCK\\_NOTIFY\\_EVENT\\_REMOVE](#) message (see the [ProcessSocketNotifications](#) and [SocketNotificationRetrieveEvents](#) topics), and then re-register the socket.

To avoid freeing memory that's still in use, you should free a registration's associated data structures only after receiving the [SOCK\\_NOTIFY\\_EVENT\\_REMOVE](#) notification for the registration. When the socket descriptor used to register for notifications is closed using the [closesocket](#) function, its notifications are automatically deregistered. However, already-queued notifications might still be delivered. An automatic deregistration via [closesocket](#) won't generate a [SOCK\\_NOTIFY\\_EVENT\\_REMOVE](#) notification.

If you want multi-threaded processing, then you should use a single I/O completion port with multiple threads processing notifications. This allows the I/O completion port to scale out the work across multiple threads, as

necessary. Avoid having multiple I/O completion ports (for example, one per thread), because that design is vulnerable to bottle-necking on a single thread while others are idle.

If multiple threads are dequeuing notification packets with level-triggered notifications, then `SOCK_NOTIFY_TRIGGER_ONESHOT` should be supplied to avoid multiple threads receiving notifications for a state change. Once the socket notification has been processed, the notification should be re-registered.

If multiple threads are dequeuing notification packets on a stream-oriented connection where individual messages need to be processed on a single thread, then consider using level-triggered one-shot notifications. That reduces the likelihood that multiple threads will receive message fragments that need to be re-assembled cross-thread.

If you're using edge-triggered notifications, then we don't recommend one-shot notifications because the socket needs to be drained after enabling registrations. This is a more complicated pattern to implement, and is more expensive because it always requires a call that returns `WSAEWOULDBLOCK`.

If you want connection acceptance scale-out on a single listening socket, then servers should use the [AcceptEx](#) function instead of subscribing to notifications for connection requests. Accepting connections in response to notifications implicitly throttles the rate of connection acceptance relative to processing requests for existing connections.

Below are code examples illustrating some socket state notification scenarios. Some of the code contains *to do* items for your own applications.

## Common code

First, here's a code listing that contains some common definitions and functions that are used by the scenarios that follow.

```
#include "pch.h"
#include <winsock2.h>
#pragma comment(lib, "Ws2_32")

#define SERVER_ADDRESS          0x0100007f // localhost
#define SERVER_PORT              0xffff      // TODO: select an actual valid port
#define MAX_TIMEOUT               1000
#define CLIENT_LOOP_COUNT        10

typedef struct SERVER_CONTEXT {
    HANDLE ioCompletionPort;
    SOCKET listenerSocket;
} SERVER_CONTEXT;

typedef struct CLIENT_CONTEXT {
    UINT32 transmitCount;
} CLIENT_CONTEXT;

SRWLOCK g_printLock = SRWLOCK_INIT;

VOID DestroyServerContext(_Inout_ _Post_invalid_ SERVER_CONTEXT* serverContext) {
    if (serverContext->listenerSocket != INVALID_SOCKET) {
        closesocket(serverContext->listenerSocket);
    }

    if (serverContext->ioCompletionPort != NULL) {
        CloseHandle(serverContext->ioCompletionPort);
    }

    free(serverContext);
}

DWORD CreateServerContext(_Outptr_ SERVER_CONTEXT** serverContext) {
    DWORD errorCode;
```

```

DWORD errorCode;
SERVER_CONTEXT* localContext = NULL;
sockaddr_in serverAddress = { };

localContext = (SERVER_CONTEXT*)malloc(sizeof(*localContext));
if (localContext == NULL) {
    errorCode = ERROR_NOT_ENOUGH_MEMORY;
    goto Exit;
}

ZeroMemory(localContext, sizeof(*localContext));
localContext->listenerSocket = INVALID_SOCKET;

localContext->ioCompletionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
if (localContext->ioCompletionPort == NULL) {
    errorCode = GetLastError();
    goto Exit;
}

localContext->listenerSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (localContext->listenerSocket == INVALID_SOCKET) {
    errorCode = GetLastError();
    goto Exit;
}

serverAddress.sin_family = AF_INET;
serverAddress.sin_addr.s_addr = SERVER_ADDRESS;
serverAddress.sin_port = SERVER_PORT;
if (bind(localContext->listenerSocket, (sockaddr*)&serverAddress, sizeof(serverAddress)) != 0) {
    errorCode = GetLastError();
    goto Exit;
}

if (listen(localContext->listenerSocket, 0) != 0) {
    errorCode = GetLastError();
    goto Exit;
}

*serverContext = localContext;
localContext = NULL;
errorCode = ERROR_SUCCESS;

Exit:
if (localContext != NULL) {
    DestroyServerContext(localContext);
}

return errorCode;
}

// Create a socket, connect to the server, send transmitCount copies of the
// payload, then disconnect.
DWORD
WINAPI
ClientThreadRoutine(_In_ PVOID clientContextPointer) {
    const UINT32 payload = 0xdeadbeef;
    CLIENT_CONTEXT* clientContext = (CLIENT_CONTEXT*)clientContextPointer;

    sockaddr_in serverAddress = {};
    SOCKET clientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (clientSocket == INVALID_SOCKET) {
        goto Exit;
    }

    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = SERVER_ADDRESS;
    serverAddress.sin_port = SERVER_PORT;
    if (connect(clientSocket, (sockaddr*)&serverAddress, sizeof(serverAddress)) != 0) {
        ...
}

```

```

        goto Exit;
    }

    for (UINT32 Index = 0; Index < clientContext->transmitCount; Index += 1) {
        if (send(clientSocket, (const char*)&payload, sizeof(payload), 0) < 0) {
            goto Exit;
        }
    }

    if (shutdown(clientSocket, SD_BOTH) != 0) {
        goto Exit;
    }

Exit:
    if (clientSocket != INVALID_SOCKET) {
        closesocket(INVALID_SOCKET);
    }

    free(clientContext);

    return 0;
}

DWORD CreateClientThread(_In_ UINT32 transmitCount) {
    DWORD errorCode = ERROR_SUCCESS;
    CLIENT_CONTEXT* clientContext = NULL;
    HANDLE clientThread = NULL;

    clientContext = (CLIENT_CONTEXT*)malloc(sizeof(*clientContext));
    if (clientContext == NULL) {
        errorCode = ERROR_NOT_ENOUGH_MEMORY;
        goto Exit;
    }

    ZeroMemory(clientContext, sizeof(*clientContext));
    clientContext->transmitCount = transmitCount;

    clientThread = CreateThread(NULL, 0, ClientThreadRoutine, clientContext, 0, NULL);
    if (clientThread == NULL) {
        errorCode = GetLastError();
        goto Exit;
    }

    clientContext = NULL;

Exit:
    if (clientContext != NULL) {
        free(clientContext);
    }

    if (clientThread != NULL) {
        CloseHandle(clientThread);
    }

    return errorCode;
}

VOID PrintError(DWORD errorCode) {
    AcquireSRWLockExclusive(&g_printLock);

    wprintf_s(L"Server thread %d encountered an error %d.", GetCurrentThreadId(), errorCode);
    WCHAR errorString[512];
    if (FormatMessageW(FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        errorCode,
        0,
        errorString,
        RTL_NUMBER_OF(errorString),
        NULL) != 0)

```

```

    {
        wprintf_s(L"%s", errorString);
    }

    ReleaseSRWLockExclusive(&g_printLock);
}

// This routine must be used only if a single socket is registered.
DWORD DeregisterAndWait(_In_ HANDLE ioCompletionPort, _In_ SOCKET socket) {
    DWORD errorCode;
    SOCK_NOTIFY_REGISTRATION registration = {};
    OVERLAPPED_ENTRY notification;
    UINT32 notificationCount;

    // Keep looping until the registration is removed, or a timeout is hit.
    while (TRUE) {

        registration.operation = SOCK_NOTIFY_OP_REMOVE;
        registration.socket = socket;
        errorCode = ProcessSocketNotifications(ioCompletionPort,
            1,
            &registration,
            MAX_TIMEOUT,
            1,
            &notification,
            &notificationCount);

        if (errorCode != ERROR_SUCCESS) {
            goto Exit;
        }

        if (registration.registrationResult != ERROR_SUCCESS) {
            errorCode = registration.registrationResult;
            goto Exit;
        }

        // Drops all non-removal notifications. Must be used only
        // if a single socket is registered.
        if (SocketNotificationRetrieveEvents(&notification) & SOCK_NOTIFY_EVENT_REMOVE) {
            break;
        }
    }

    Exit:
    return errorCode;
}

```

## Simple replacement for polling

This scenario demonstrates a drop-in replacement for applications using poll ([WSAPoll](#)) or similar APIs. It's single-threaded, and makes use of persistent (not one-shot) registrations. Because the registration doesn't need to be re-registered, it uses [GetQueuedCompletionStatusEx](#) to dequeue notifications.

```

VOID SimplePollReplacement() {
    DWORD errorCode;
    WSADATA wsaData;
    SERVER_CONTEXT* serverContext = NULL;
    SOCKET tcpAcceptSocket = INVALID_SOCKET;
    u_long nonBlocking = 1;
    SOCKET currentSocket;
    SOCK_NOTIFY_REGISTRATION registration = {};
    OVERLAPPED_ENTRY notification;
    ULONG notificationCount;
    UINT32 events;
    CHAR dataBuffer[512];

```

```

if (WSAStartup(WINSOCK_VERSION, &wsaData) != 0) {
    errorCode = GetLastError();
    PrintError(errorCode);
    return;
}

errorCode = CreateServerContext(&serverContext);
if (errorCode != ERROR_SUCCESS) {
    goto Exit;
}

errorCode = CreateClientThread(CLIENT_LOOP_COUNT);
if (errorCode != ERROR_SUCCESS) {
    goto Exit;
}

tcpAcceptSocket = accept(serverContext->listenerSocket, NULL, NULL);
if (tcpAcceptSocket == INVALID_SOCKET) {
    errorCode = GetLastError();
    goto Exit;
}

if (ioctlsocket(tcpAcceptSocket, FIONBIO, &nonBlocking) != 0) {
    errorCode = GetLastError();
    goto Exit;
}

// Register the accepted connection.
registration.completionKey = (PVOID)tcpAcceptSocket;
registration.eventFilter = SOCK_NOTIFY_REGISTER_EVENT_IN | SOCK_NOTIFY_REGISTER_EVENT_HANGUP;
registration.operation = SOCK_NOTIFY_OP_ENABLE;
registration.triggerFlags = SOCK_NOTIFY_TRIGGER_LEVEL;
registration.socket = tcpAcceptSocket;
errorCode = ProcessSocketNotifications(serverContext->ioCompletionPort,
    1,
    &registration,
    0,
    0,
    NULL,
    NULL);

// Make sure all registrations were processed.
if (errorCode != ERROR_SUCCESS) {
    goto Exit;
}

// Make sure each registration was successful.
if (registration.registrationResult != ERROR_SUCCESS) {
    errorCode = registration.registrationResult;
    goto Exit;
}

// Keep receiving data until the client disconnects.
while (TRUE) {

    wprintf_s(L"Waiting for client action...\r\n");

    if (!GetQueuedCompletionStatusEx(serverContext->ioCompletionPort,
        &notification,
        1,
        &notificationCount,
        MAX_TIMEOUT,
        FALSE))
    {
        errorCode = GetLastError();
        goto Exit;
    }
}

```

```

// The completion key is the socket we supplied above.
//
// This is true only because the registration supplied the socket as the completion
// key. A more typical pattern is to supply a context pointer. This example supplies
// the socket directly, for simplicity.
//
// The events are stored in the number-of-bytes-received field.
events = SocketNotificationRetrieveEvents(&notification);

currentSocket = (SOCKET)notification.lpCompletionKey;
if (events & SOCK_NOTIFY_EVENT_IN) {

    // We don't check for a 0-size receive because we subscribed to hang-up notifications.
    if (recv(currentSocket, dataBuffer, sizeof(dataBuffer), 0) < 0) {
        errorCode = GetLastError();
        goto Exit;
    }

    wprintf_s(L"Received client data.\r\n");
}

if (events & SOCK_NOTIFY_EVENT_HANGUP) {
    wprintf_s(L"Client hung up. Exiting. \r\n");
    break;
}

if (events & SOCK_NOTIFY_EVENT_ERR) {
    wprintf_s(L"The socket was ungracefully reset or another error occurred. Exiting.\r\n");
    // Obtain a more detailed error code by issuing a non-blocking receive.
    recv(currentSocket, dataBuffer, sizeof(dataBuffer), 0);
    errorCode = GetLastError();
    goto Exit;
}
}

errorCode = ERROR_SUCCESS;

Exit:
if (errorCode != ERROR_SUCCESS) {
    PrintError(errorCode);
}

if (serverContext != NULL) {
    if (tcpAcceptSocket != INVALID_SOCKET) {
        DeregisterAndWait(serverContext->ioCompletionPort, tcpAcceptSocket);
    }

    DestroyServerContext(serverContext);
}

if (tcpAcceptSocket != INVALID_SOCKET) {
    closesocket(tcpAcceptSocket);
}

WSACleanup();
}

```

## Edge-triggered UDP server

This is a simple illustration of how to use the APIs with edge-triggering.

### IMPORTANT

The server must keep receiving until it receives a **WSAEWOULDBLOCK**. Otherwise, it can't be sure that a rising edge will be observed. As such, the server's socket must also be non-blocking.

This example uses UDP to demonstrate the lack of a **HANGUP** notification. It takes some liberties with assuming the common helpers create UDP sockets if needed.

```
// This example assumes that substantially similar helpers are available for UDP sockets.
VOID SimpleEdgeTriggeredSample() {
    DWORD errorCode;
    WSADATA wsaData;
    SOCKET serverSocket = INVALID_SOCKET;
    SOCKET currentSocket;
    HANDLE ioCompletionPort = NULL;
    sockaddr_in serverAddress = { };
    u_long nonBlocking = 1;
    SOCK_NOTIFY_REGISTRATION registration = {};
    OVERLAPPED_ENTRY notification;
    ULONG notificationCount;
    UINT32 events;
    CHAR dataBuffer[512];
    UINT32 datagramCount;
    int receiveResult;

    if (WSAStartup(WINSOCK_VERSION, &wsaData) != 0) {
        errorCode = GetLastError();
        PrintError(errorCode);
        return;
    }

    ioCompletionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
    if (ioCompletionPort == NULL) {
        errorCode = GetLastError();
        goto Exit;
    }

    serverSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (serverSocket == INVALID_SOCKET) {
        errorCode = GetLastError();
        goto Exit;
    }

    // Register the server UDP socket before binding to a port to ensure data doesn't become
    // present before the registration. Otherwise, the server could miss the notification and
    // hang.
    //
    // Edge-triggered is not recommended with one-shot due to the difficulty in re-registering.
    registration.completionKey = (PVOID)serverSocket;
    registration.eventFilter = SOCK_NOTIFY_EVENT_IN;
    registration.operation = SOCK_NOTIFY_OP_ENABLE;
    registration.triggerFlags = SOCK_NOTIFY_TRIGGER_EDGE;
    registration.socket = serverSocket;
    errorCode = ProcessSocketNotifications(ioCompletionPort, 1, &registration, 0, 0, NULL, NULL);
    if (errorCode != ERROR_SUCCESS) {
        goto Exit;
    }

    if (registration.registrationResult != ERROR_SUCCESS) {
        errorCode = registration.registrationResult;
        goto Exit;
    }

    // Use non-blocking sockets with edge-triggered notifications, since the data must be
    // drained before a rising edge can be observed again.
    errorCode = ioctlsocket(serverSocket, FIONBIO, &nonBlocking);
    if (errorCode != ERROR_SUCCESS) {
        goto Exit;
    }

    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = SERVER_ADDRESS;
    //-----Add-----the-----port-----here-----for-----the-----server-----port-----
```

```

serverAddress.sin_port = SERVER_PORT;
if (bind(serverSocket, (sockaddr*)&serverAddress, sizeof(serverAddress)) != 0) {
    errorCode = GetLastError();
    goto Exit;
}

// Create the client.
// While CreateClientThread connects to a TCP socket and sends data over it, for this example
// assume that CreateClientThread creates a UDP socket instead, and sends data over it.
errorCode = CreateClientThread(CLIENT_LOOP_COUNT);
if (errorCode != ERROR_SUCCESS) {
    goto Exit;
}

// Receive the packets.
datagramCount = 0;
while (datagramCount < CLIENT_LOOP_COUNT) {

    wprintf_s(L"Waiting for client action...\r\n");

    if (!GetQueuedCompletionStatusEx(ioCompletionPort,
        &notification,
        1,
        &notificationCount,
        MAX_TIMEOUT,
        FALSE))
    {
        errorCode = GetLastError();
        goto Exit;
    }

    // The completion key is the socket we supplied above.
    //
    // This is true only because the registration supplied the socket as the completion
    // key. A more typical pattern is to supply a context pointer. This example supplies
    // the socket directly, for simplicity.
    //
    // The events are the integer value of the overlapped pointer.
    events = SocketNotificationRetrieveEvents(&notification);

    currentSocket = (SOCKET)notification.lpCompletionKey;
    if (events & SOCK_NOTIFY_EVENT_ERR) {
        // Obtain a more detailed error code by issuing a non-blocking receive.
        recv(currentSocket, dataBuffer, sizeof(dataBuffer), 0);
        errorCode = GetLastError();
        goto Exit;
    }

    if ((events & SOCK_NOTIFY_EVENT_IN) == 0) {
        continue;
    }

    // Keep looping receiving data until the read would block, otherwise the edge may not
    // have been reset.
    while (TRUE) {
        receiveResult = recv(currentSocket, dataBuffer, sizeof(dataBuffer), 0);
        if (receiveResult < 0) {
            errorCode = GetLastError();
            if (errorCode != WSAEWOULDBLOCK) {
                goto Exit;
            }
        }

        break;
    }

    datagramCount += 1;
    wprintf_s(L"Received client data.\r\n");
}
}

```

```

wprintf_s(L"Received all data. Exiting... \r\n");
errorCode = ERROR_SUCCESS;

Exit:
if (errorCode != ERROR_SUCCESS) {
    PrintError(errorCode);
}

if (serverSocket != INVALID_SOCKET) {
    if (ioCompletionPort != NULL) {
        DeregisterAndWait(ioCompletionPort, serverSocket);
    }

    closesocket(serverSocket);
}

if (ioCompletionPort != NULL) {
    CloseHandle(ioCompletionPort);
}

WSACleanup();
}

```

## Multi-threaded server

This example demonstrates a more realistic multi-threaded use pattern that uses the I/O completion port's scale-out capabilities to distribute work across multiple server threads. The server uses one-shot level-triggering to avoid multiple threads picking up notifications for the same socket, and to allow each thread to drain received data one chunk at a time.

It also demonstrates some common patterns used with the completion port. The completion key is used to supply a per-socket context pointer. The context pointer has a header that describes the type of socket being used, so that multiple sockets types can be used on a single completion port. Comments in the example highlight that arbitrary completions can be dequeued (just as with the [GetQueuedCompletionStatusEx](#) function), not only socket notifications. The [PostQueuedCompletionStatus](#) API is used to post messages to threads, and wake them without having to wait for the arrival of a socket notification.

Finally, the example demonstrates some of the intricacies of correctly deregistering and cleaning up socket contexts in a threaded workload. In this example, socket context is implicitly owned by the thread that receives the notification. The thread maintains ownership if it fails to register the notification.

```

#define CLIENT_THREAD_COUNT          100
// The I/O completion port infrastructure ensures that the system isn't over-subscribed by
// ensuring server-side threads block if they exceed the number of logical processors. If the
// machine has more than 16 logical processors, then this can be observed by increasing this number.
#define SERVER_THREAD_COUNT         16
#define SERVER_DEQUEUE_COUNT        3
#define SERVER_EXIT_KEY             ((ULONG_PTR)-1)

typedef struct SERVER_THREAD_CONTEXT {
    SERVER_CONTEXT* commonContext;
    SRWLOCK stateLock;
    _Guarded_by_(stateLock) UINT32 deregisterCount;
    _Guarded_by_(stateLock) BOOLEAN shouldExit;
} SERVER_THREAD_CONTEXT;

typedef enum SOCKET_TYPE {
    SOCKET_TYPE_LISTENER,
    SOCKET_TYPE_ACCEPT
} SOCKET_TYPE;

typedef struct SOCKET_CONTEXT {

```

```

    SOCKET_TYPE socketType;
    SOCKET socket;
} SOCKET_CONTEXT;

VOID CancelServerThreadsAsync(_Inout_ SERVER_THREAD_CONTEXT* serverThreadContext) {
    AcquireSRWLockExclusive(&serverThreadContext->stateLock);
    serverThreadContext->shouldExit = TRUE;
    ReleaseSRWLockExclusive(&serverThreadContext->stateLock);
}

VOID IndicateServerThreadExit(_In_ HANDLE ioCompletionPort) {
    // Notify a server thread that it needs to exit. It can then notify the other threads when it
    // exits.
    //
    // If this fails, then server threads may hang, and this program will never terminate. That
    // is an unrecoverable error.
    if (!PostQueuedCompletionStatus(ioCompletionPort, 0, SERVER_EXIT_KEY, NULL)) {
        RaiseFailFastException(NULL, NULL, 0);
    }
}

VOID DestroySocketContext(_Inout_ _Post_invalid_ SOCKET_CONTEXT* socketContext) {
    if (socketContext->socket != INVALID_SOCKET) {
        closesocket(socketContext->socket);
    }

    free(socketContext);
}

DWORD AcceptConnection(_In_ SOCKET listenSocket, _Outptr_ SOCKET_CONTEXT** socketContextOut) {
    DWORD errorCode;
    SOCKET_CONTEXT* socketContext = NULL;

    socketContext = (SOCKET_CONTEXT*)malloc(sizeof(*socketContext));
    if (socketContext == NULL) {
        errorCode = ERROR_NOT_ENOUGH_MEMORY;
        goto Exit;
    }

    ZeroMemory(socketContext, sizeof(*socketContext));
    socketContext->socketType = SOCKET_TYPE_ACCEPT;
    socketContext->socket = accept(listenSocket, NULL, NULL);
    if (socketContext->socket == INVALID_SOCKET) {
        errorCode = GetLastError();
        goto Exit;
    }

    *socketContextOut = socketContext;
    socketContext = NULL;

Exit:
    if (socketContext != NULL) {
        _ASSERT(errorCode != ERROR_SUCCESS);
        DestroySocketContext(socketContext);
    }

    return errorCode;
}

DWORD
WINAPI
ServerThreadRoutine(_In_ PVOID serverThreadContextPointer) {
    DWORD errorCode;
    SERVER_THREAD_CONTEXT* serverThreadContext;
    HANDLE ioCompletionPort;
    // Accepting a connection requires two registrations: one to re-enable the listening socket
    // notification, and one to register the newly-accepted connection.
    SOCK_NOTIFY_REGISTRATION registrationBuffer[SERVER_DEQUEUE_COUNT * 2];
    UINT32 registrationCount;
}

```

```

SOCK_NOTIFY_REGISTRATION* registration;
OVERLAPPED_ENTRY notifications[SERVER_DEQUEUE_COUNT];
UINT32 notificationCount;
UINT32 events;
SOCKET_CONTEXT* socketContext;
SOCKET_CONTEXT* acceptedContext;
BOOLEAN shouldExit;
CHAR dataBuffer[512];

serverThreadContext = (SERVER_THREAD_CONTEXT*)serverThreadContextPointer;
ioCompletionPort = serverThreadContext->commonContext->ioCompletionPort;

// Boot-strap the loop process.
registrationCount = 0;

// Keep looping, processing notifications until exit has been requested.
while (TRUE) {

    AcquireSRWLockExclusive(&serverThreadContext->stateLock);
    shouldExit = serverThreadContext->shouldExit;
    ReleaseSRWLockExclusive(&serverThreadContext->stateLock);
    if (shouldExit) {
        goto Exit;
    }

    AcquireSRWLockExclusive(&g_printLock);
    wprintf_s(L"Server thread %d waiting for client action...\\r\\n", GetCurrentThreadId());
    ReleaseSRWLockExclusive(&g_printLock);

    // Process notifications and re-register one-shot notifications that were processed on a
    // previous iteration.
    errorCode = ProcessSocketNotifications(ioCompletionPort,
        registrationCount,
        (registrationCount == 0) ? NULL : registrationBuffer,
        MAX_TIMEOUT,
        RTL_NUMBER_OF(notifications),
        notifications,
        &notificationCount);

    // TODO: Production code should handle failure better. This can fail due to transient memory
    conditions, or due to
    // invalid input such as a bad handle. Retrying in case the memory conditions abate is
    // a reasonable strategy.
    if (errorCode != ERROR_SUCCESS) {
        goto Exit;
    }

    // Check whether any registrations failed, and attempt to clean up if they did.
    errorCode = ERROR_SUCCESS;
    for (UINT32 i = 0; i < registrationCount; i += 1) {
        registration = &registrationBuffer[i];
        if (registration->registrationResult == ERROR_SUCCESS) {
            continue;
        }

        // Preserve the first failure code.
        if (errorCode == ERROR_SUCCESS) {
            errorCode = registration->registrationResult;
        }

        // All the registrations are oneshot, so if the registration failed, then only this thread
        // has access to the context. Attempt to clean up fully:
        // - The listening socket is owned by the main thread, so ignore that.
        // - If the socket hasn't been registered, just free its memory.
        // - Otherwise, attempt to deregister it.

        socketContext = (SOCKET_CONTEXT*)registration->completionKey;
        if (socketContext->socketType == SOCKET_TYPE_LISTENER) {
            continue;
        }
    }
}

```

```

        }

        // Best-effort de-registration. In case of failure, simply get rid of the socket and
        // context. This is safe to do because the notification for the socket can't be enabled.
        // Either it was never registered in the first place, or re-registration failed, and it
        // was previously disabled by nature of being a one-shot registration.
        registration->operation = SOCK_NOTIFY_OP_REMOVE;
        errorCode = ProcessSocketNotifications(ioCompletionPort,
            1,
            registration,
            0,
            0,
            NULL,
            NULL);

        if ((errorCode != ERROR_SUCCESS) ||
            (registration->registrationResult != ERROR_SUCCESS)) {
            DestroySocketContext(socketContext);
        }
    }

    // Process the notifications. Many will need to be re-enabled because they are one-shot,
    // so ensure that we can build that incrementally.
    registrationCount = 0;
    ZeroMemory(registrationBuffer, sizeof(registrationBuffer));

    for (UINT32 i = 0; i < notificationCount; i += 1) {
        if (notifications[i].lpCompletionKey == SERVER_EXIT_KEY) {
            _ASSERT(serverThreadContext->shouldExit);

            // On exit, this thread will post the next exit message.
            errorCode = ERROR_SUCCESS;
            goto Exit;
        }

        socketContext = (SOCKET_CONTEXT*)notifications[i].lpCompletionKey;
        events = SocketNotificationRetrieveEvents(&notifications[i]);

        // Process the socket notification, taking socket-specific actions.
        switch (socketContext->socketType) {
        case SOCKET_TYPE_LISTENER:

            // Accepting connections in response to notifications implicitly throttles
            // the rate at which incoming connections are accepted, and limits scale-out for
            // new connection acceptance. Consider using AcceptEx if greater scaling of
            // connection acceptance is desired.

            // Perform an accept regardless of the notification. The only possible notifications
            // are for available connections or error conditions. Any possible error conditions
            // will be processed as part of the accept.
            errorCode = AcceptConnection(socketContext->socket, &acceptedContext);
            if (errorCode == ERROR_SUCCESS) {
                // Register the accepted connection.
                registration = &registrationBuffer[registrationCount];
                registration->socket = acceptedContext->socket;
                registration->completionKey = acceptedContext;
                registration->eventFilter = SOCK_NOTIFY_EVENT_IN | SOCK_NOTIFY_EVENT_HANGUP;
                registration->operation =
                    SOCK_NOTIFY_OP_ENABLE;
                registration->triggerFlags = SOCK_NOTIFY_TRIGGER_ONESHOT | SOCK_NOTIFY_TRIGGER_LEVEL;
                registrationCount += 1;
            }

            // Re-arm the existing listening socket registration.
            registration = &registrationBuffer[registrationCount];
            registration->socket = socketContext->socket;
            registration->completionKey = socketContext;
            registration->eventFilter = SOCK_NOTIFY_EVENT_IN;
            registration->operation =

```

```

registration->operation =
    SOCK_NOTIFY_OP_ENABLE;
registration->triggerFlags = SOCK_NOTIFY_TRIGGER_ONESHOT | SOCK_NOTIFY_TRIGGER_LEVEL;
registrationCount += 1;
break;

case SOCKET_TYPE_ACCEPT:
    // The registration was removed. Clean up the context.
    if (events & SOCK_NOTIFY_EVENT_REMOVE) {
        AcquireSRWLockExclusive(&serverThreadContext->stateLock);
        serverThreadContext->deregisterCount += 1;
        if (serverThreadContext->deregisterCount >= CLIENT_THREAD_COUNT) {
            serverThreadContext->shouldExit = TRUE;
        }
        ReleaseSRWLockExclusive(&serverThreadContext->stateLock);

        DestroySocketContext(socketContext);
        continue;
    }

registration = &registrationBuffer[registrationCount];

// If a hangup occurred, then remove the registration.
if (events & SOCK_NOTIFY_EVENT_HANGUP) {
    registration->eventFilter = 0;
    registration->operation = SOCK_NOTIFY_OP_REMOVE;
}

// Receive data.
if (events & (SOCK_NOTIFY_EVENT_IN | SOCK_NOTIFY_EVENT_ERR)) {
    // TODO: Handle errors (for example, due to connection reset). The error from recv can
    // be used to retrieve the underlying socket for a SOCK_NOTIFY_EVENT_ERR.
    if (recv(socketContext->socket, dataBuffer, sizeof(dataBuffer), 0) < 0) {
        registration->operation = SOCK_NOTIFY_OP_REMOVE;
        registration->eventFilter = 0;
    }
    else {
        registration->operation |=
            SOCK_NOTIFY_OP_ENABLE;
        registration->triggerFlags =
            SOCK_NOTIFY_TRIGGER_ONESHOT | SOCK_NOTIFY_TRIGGER_LEVEL;
        registration->eventFilter = SOCK_NOTIFY_EVENT_IN | SOCK_NOTIFY_EVENT_HANGUP;
    }
}

registration->socket = socketContext->socket;
registration->completionKey = socketContext;
registrationCount += 1;
break;

// TODO:
//
// Other (potentially non-socket) I/O completion can be processed here. For instance,
// this could also be processing disk I/O. The contexts will need to have a common
// header that can be used to differentiate between the different context types,
// similar to how the listening and accepted sockets are differentiated.
//
// case ... :

default:
    _ASSERT(!"Unexpected socket type!");
    errorCode = ERROR_UNIDENTIFIED_ERROR;
    goto Exit;
}
}

errorCode = ERROR_SUCCESS;

```

```

exit:
    // If an error occurred, then ensure the other threads know they should exit.
    // TODO: use an error handling strategy that isn't just exiting.
    if (errorCode != ERROR_SUCCESS) {
        PrintError(errorCode);
        CancelServerThreadsAsync(serverThreadContext);
    }

    // Wake a remaining server thread.
    IndicateServerThreadExit(ioCompletionPort);

    AcquireSRWLockExclusive(&g_printLock);
    wprintf_s(L"Server thread %d exited\r\n", GetCurrentThreadId());
    ReleaseSRWLockExclusive(&g_printLock);

    return errorCode;
}

VOID MultiThreadedTcpServer() {
    DWORD errorCode;
    WSADATA wsaData;
    SERVER_THREAD_CONTEXT serverContext = { NULL, SRWLOCK_INIT, 0, FALSE };
    SOCKET_CONTEXT listenContext = {};
    SOCK_NOTIFY_REGISTRATION registration = {};
    HANDLE serverThreads[SERVER_THREAD_COUNT] = {};
    UINT32 serverThreadCount = 0;

    if (WSAStartup(WINSOCK_VERSION, &wsaData) != 0) {
        errorCode = GetLastError();
        PrintError(errorCode);
        return;
    }

    listenContext.socket = INVALID_SOCKET;
    listenContext.socketType = SOCKET_TYPE_LISTENER;
    errorCode = CreateServerContext(&serverContext.commonContext);
    if (errorCode != ERROR_SUCCESS) {
        goto Exit;
    }

    // Register the listening socket with the I/O completion port so the server threads are notified
    // of incoming connections.
    listenContext.socket = serverContext.commonContext->listenerSocket;
    registration.completionKey = &listenContext;
    registration.eventFilter = SOCK_NOTIFY_EVENT_IN;
    registration.operation = SOCK_NOTIFY_OP_ENABLE;
    registration.triggerFlags = SOCK_NOTIFY_TRIGGER_LEVEL | SOCK_NOTIFY_TRIGGER_PERSISTENT;
    registration.socket = listenContext.socket;
    errorCode = ProcessSocketNotifications(serverContext.commonContext->ioCompletionPort,
                                           1,
                                           &registration,
                                           0,
                                           0,
                                           NULL,
                                           NULL);

    if (errorCode != ERROR_SUCCESS) {
        goto Exit;
    }

    // Create the server threads. These are likely over-subscribed, but the I/O completion port
    // ensures that they scale appropriately.
    while (serverThreadCount < RTL_NUMBER_OF(serverThreads)) {
        serverThreads[serverThreadCount] =
            CreateThread(NULL, 0, ServerThreadRoutine, &serverContext, 0, NULL);

        if (serverThreads[serverThreadCount] == NULL) {
            errorCode = GetLastError();
            goto Exit;
        }
    }
}

```

```

    }

    // Create the client threads, which are badly over-subscribed.
    for (UINT32 i = 0; i < CLIENT_THREAD_COUNT; i += 1) {
        errorCode = CreateClientThread(CLIENT_LOOP_COUNT);
        if (errorCode != ERROR_SUCCESS) {
            goto Exit;
        }
    }

    errorCode = ERROR_SUCCESS;

Exit:
    if (errorCode != ERROR_SUCCESS) {
        PrintError(errorCode);

        // In case of error, ensure that all server threads know to exit.
        if (serverContext.commonContext != NULL) {
            CancelServerThreadsAsync(&serverContext);
            IndicateServerThreadExit(serverContext.commonContext->ioCompletionPort);
        }
    }

    if (serverThreadCount > 0) {
        wprintf_s(L"Waiting for %d server threads to exit...\r\n", serverThreadCount);
        errorCode = WaitForMultipleObjects(serverThreadCount, serverThreads, TRUE, INFINITE);
        _ASSERT(errorCode == ERROR_SUCCESS);
    }

    // TODO: In case of failure, clean up remaining state. For example, Accepted connections can be kept in
    // a global list, which can be closed from this thread.

    for (UINT32 i = 0; i < serverThreadCount; i += 1) {
        CloseHandle(serverThreads[i]);
    }

    DestroyServerContext(serverContext.commonContext);
    WSACleanup();
}

```

## See also

- [ProcessSocketNotifications](#)
- [SOCK\\_NOTIFY\\_REGISTRATION](#)
- [SocketNotificationRetrieveEvents](#)

# Winsock Reference

3/5/2021 • 2 minutes to read • [Edit Online](#)

Winsock reference is divided into the following sections:

- [Socket Options](#)
- [Winsock IOCTLs](#)
- [Winsock Annexes](#)
- [Winsock Enumerations](#)
- [Winsock Functions](#)
- [Winsock Structures](#)
- [Winsock SPI](#)
- [Windows Sockets Error Codes](#)

## Related topics

[About Winsock](#)

[Using Winsock](#)

[What's New for Windows Sockets](#)

[Winsock Network Protocol Support in Windows](#)

# Socket Options

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section describes Winsock Socket Options for various editions of Windows operating systems. Use the [getsockopt](#) and [setsockopt](#) functions for more getting and setting socket options. To enumerate protocols and discover supported properties for each installed protocol, use the [WSAEnumProtocols](#) function.

Some socket options require more explanation than these tables can convey; such options contain links to additional pages.

## IPPROTO\_IP

Socket options applicable at the IPv4 level. For more information, see the [IPPROTO\\_IP Socket Options](#).

## IPPROTO\_IPV6

Socket options applicable at the IPv6 level. For more information, see the [IPPROTO\\_IPV6 Socket Options](#).

## IPPROTO\_RM

Socket options applicable at the reliable multicast level. For more information, see the [IPPROTO\\_RM Socket Options](#).

## IPPROTO\_TCP

Socket options applicable at the TCP level. For more information, see the [IPPROTO\\_TCP Socket Options](#).

## IPPROTO\_UDP

Socket options applicable at the UDP level. For more information, see the [IPPROTO\\_UDP Socket Options](#).

## NSPROTO\_IPX

Socket options applicable at the IPX level. For more information, see the [NSPROTO\\_IPX Socket Options](#).

## SOL\_APPLETALK

Socket options applicable at the AppleTalk level. For more information, see the [SOL\\_APPLETALK Socket Options](#).

## SOL\_IRLMP

Socket options applicable at the InfraRed Link Management Protocol level. For more information, see the [SOL\\_IRLMP Socket Options](#).

## SOL\_SOCKET

Socket options applicable at the socket level. For more information, see the [SOL\\_SOCKET Socket Options](#).

## Remarks

All SO\_\* socket options apply equally to IPv4 and IPv6 (except SO\_BROADCAST, since broadcast is not implemented in IPv6).



# IPPROTO\_IP socket options

3/22/2021 • 10 minutes to read • [Edit Online](#)

The following tables describe IPPROTO\_IP socket options that apply to sockets created for the IPv4 address family (AF\_INET). See the [getsockopt](#) and [setsockopt](#) function reference pages for more information on getting and setting socket options.

To enumerate protocols and discover supported properties for each installed protocol, use the [WSAEnumProtocols](#), [WSCEnumProtocols](#), or [WSCEnumProtocols32](#) function.

Some socket options require more explanation than these tables can convey; such options contain links to additional pages.

## Options

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IP_ADD_IFLIST		yes	DWORD (IF_INDEX)	Adds an interface index to the IFLIST associated with the <a href="#">IP_IFLIST</a> option.
IP_ADD_MEMBERSHIP		yes	<a href="#">ip_mreq</a>	Join the socket to the supplied multicast group on the specified interface.
IP_ADD_SOURCE_MEMBERSHIP		yes	<a href="#">ip_mreq_source</a>	Join the supplied multicast group on the given interface and accept data sourced from the supplied source address.
IP_BLOCK_SOURCE		yes	<a href="#">ip_mreq_source</a>	Removes the given source as a sender to the supplied multicast group and interface.
IP_DEL_IFLIST		yes	DWORD (IF_INDEX)	Removes an interface index from the IFLIST associated with the <a href="#">IP_IFLIST</a> option. Entries can be removed only by the application, so be aware that entries might go stale once an interface is removed.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IP_DONTFRAGMENT	yes	yes	DWORD (boolean)	Indicates that data should not be fragmented regardless of the local MTU. Valid only for message oriented protocols. Microsoft TCP/IP providers respect this option for UDP and ICMP.
IP_DROP_MEMBERS HIP		yes	<a href="#">ip_mreq</a>	Leaves the specified multicast group from the specified interface. Service providers must support this option when multicast is supported. Support is indicated in the <a href="#">WSAPROTOCOL_INF</a> structure returned by a <a href="#">WSAEnumProtocols</a> function call with the following: XPI_SUPPORT_MULTIPOINT=1, XP1_MULTIPOINT_CONTROL_PLANE=0, XP1_MULTIPOINT_DEFAULT_PLANE=0.
IP_DROP_SOURCE_MEMBERSHIP		yes	<a href="#">ip_mreq_source</a>	Drops membership to the given multicast group, interface, and source address.
IP_GET_IFLIST	yes		DWORD[] (IF_INDEX[])	Gets the current IFLIST associated with the IP_IFLIST option. Returns error if IP_IFLIST is not enabled.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IP_HDRINCL	yes	yes	DWORD (boolean)	When set to TRUE, indicates the application provides the IP header. Applies only to SOCK_RAW sockets. The TCP/IP service provider may set the ID field, if the value supplied by the application is zero. The IP_HDRINCL option is applied only to the SOCK_RAW type of protocol. A TCP/IP service provider that supports SOCK_RAW should also support IP_HDRINCL.
IP_IIFLIST	yes	yes	DWORD (boolean)	Gets or sets the IP_IIFLIST state of the socket. When this option is set to true, Datagram reception is restricted to interfaces that are in the IIFLIST. Datagrams received on any other interfaces are ignored. IIFLIST starts empty. Use IP_ADD_IIFLIST and IP_DEL_IIFLIST to edit the IIFLIST.
IP_MTU	yes		DWORD	Gets the system's estimate of the path MTU. Socket must be connected.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IP_MTU_DISCOVER	yes	yes	DWORD (PMTUD_STATE)	<p>Gets or sets the path MTU discovery state for the socket. The default value is <b>IP_PMTUDISC_NOT_SET</b>. For stream sockets, <b>IP_PMTUDISC_NOT_SET</b> and <b>IP_PMTUDISC_DONE</b> will perform path MTU discovery. <b>IP_PMTUDISC_DONE</b> and <b>IP_PMTUDISC_PRobe</b> will turn off path MTU discovery. For datagram sockets, <b>IP_PMTUDISC_DONE</b> will force all outgoing packets to have the DF bit set and an attempt to send packets larger than path MTU will result in an error. <b>IP_PMTUDISC_DONE</b> will force all outgoing packets to have the DF bit not set, and packets will be fragmented according to interface MTU. <b>IP_PMTUDISC_PRobe</b> will force all outgoing packets to have the DF bit set, and an attempt to send packets larger than interface MTU will result in an error.</p>

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IP_MULTICAST_IF	yes	yes	DWORD	<p>Gets or sets the outgoing interface for sending IPv4 multicast traffic. This option does not change the default interface for receiving IPv4 multicast traffic. The input value for setting this option is a 4-byte IPv4 address in network byte order. This DWORD parameter can also be an interface index in network byte order. Any IP address in the 0.x.x.x block (first octet of 0) except IPv4 address 0.0.0.0 is treated as an interface index. An interface index is a 24-bit number, and the 0.0.0.0/8 IPv4 address block is not used (this range is reserved). The interface index can be used to specify the default interface for multicast traffic for IPv4. If <i>optval</i> is zero, the default interface for receiving multicast is specified for sending multicast traffic. When getting this option, the <i>optval</i> returns the current default interface index for sending multicast IPv4 traffic in host byte order.</p>

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IP_MULTICAST_LOOP	yes	yes	DWORD (boolean)	Controls whether data sent by an application on the local computer (not necessarily by the same socket) in a multicast session will be received by a socket joined to the multicast destination group on the loopback interface. A value of <b>TRUE</b> causes multicast data sent by an application on the local computer to be delivered to a listening socket on the loopback interface. A value of <b>FALSE</b> prevents multicast data sent by an application on the local computer from being delivered to a listening socket on the loopback interface. By default, <b>IP_MULTICAST_LOOPBACK</b> is enabled.
IP_MULTICAST_TTL	yes	yes	DWORD	Sets/gets the TTL value associated with IP multicast traffic on the socket.
IP_OPTIONS	yes	yes	char []	Specifies IP options to be inserted into outgoing packets. Setting new options overwrites all previously specified options. Setting optval to zero removes all previously specified options. IP_OPTIONS support is not required; to check whether IP_OPTIONS is supported, use <a href="#">getsockopt</a> to get current options. If <a href="#">getsockopt</a> fails, IP_OPTIONS is not supported.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IP_ORIGINAL_ARRIVAL_IF	yes	yes	DWORD (boolean)	Indicates if the <a href="#">LPFN_WSARECVM SG (WSARecvMsg)</a> function should return optional control data containing the arrival interface where the packet was received for datagram sockets. This option allows the IPv4 interface where the packet was received to be returned in the <a href="#">WSAMSG</a> structure. This option is only valid on datagram and raw sockets (the socket type must be SOCK_DGRAM or SOCK_RAW).
IP_PKTINFO	yes	yes	DWORD	Indicates that packet information should be returned by the <a href="#">WSARecvMsg</a> function.
IP_RECEIVE_BROADCAST	yes	yes	DWORD (boolean)	Allows or blocks broadcast reception.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IP_RECVIF	yes	yes	DWORD (boolean)	<p>Indicates whether the IP stack should populate the control buffer with details about which interface received a packet with a datagram socket. When this value is true, the <a href="#">LPFN_WSARECVM SG (WSARecvMsg)</a> function will return optional control data containing the interface where the packet was received for datagram sockets. This option allows the IPv4 interface where the packet was received to be returned in the <a href="#">WSAMSG</a> structure. This option is only valid on datagram and raw sockets (the socket type must be SOCK_DGRAM or SOCK_RAW).</p>

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IP_RECVTOS	yes	yes	DWORD (boolean)	<p>Indicates whether the IP stack should populate the control buffer with a message containing the Type of Service (TOS) IPv4 header field on a received datagram. When this value is true, the <a href="#">LPFN_WSARECVM SG (WSARecvMsg)</a> function will return optional control data containing the TOS IPv4 header field value of the received datagram. This option allows the TOS IPv4 header field of the received datagram to be returned in the <a href="#">WSAMSG</a> structure. The returned message type will be IP_TOS. All DSCP and ECN bits of the TOS field will be returned. This option is only valid on datagram sockets (the socket type must be SOCK_DGRAM).</p>
IP_RECVTTL	yes	yes	DWORD (boolean)	<p>Indicates that hop (TTL) information should be returned in the <a href="#">LPFN_WSARECVM SG (WSARecvMsg)</a> function. If <i>optval</i> is set to 1 on the call to <a href="#">setsockopt</a>, the option is enabled. If set to 0, the option is disabled. This option is only valid for datagram and raw sockets (the socket type must be SOCK_DGRAM or SOCK_RAW).</p>

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IP_TOS	yes	yes	DWORD (boolean)	Do not use. Type of Service (TOS) settings should only be set using the Quality of Service API. See <a href="#">Differentiated Services</a> in the Quality of Service section of the Platform SDK for more information.
IP_TTL	yes	yes	DWORD (boolean)	Changes the default value set by the TCP/IP service provider in the TTL field of the IP header in outgoing datagrams. IP_TTL support is not required; to check whether IP_TTL is supported, use <a href="#">getsockopt</a> to get current options. If <a href="#">getsockopt</a> fails, IP_TTL is not supported.
IP_UNBLOCK_SOURCE		yes	<a href="#">ip_mreq_source</a>	Adds the given source as a sender to the supplied multicast group and interface.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IP_UNICAST_IF	yes	yes	DWORD (IF_INDEX)	<p>Gets or sets the outgoing interface for sending IPv4 traffic. This option does not change the default interface for receiving IPv4 traffic. This option is important for multihomed computers. The input value for setting this option is a 4-byte IPv4 address in network byte order. This DWORD parameter must be an interface index in network byte order. Any IP address in the 0.x.x.x block (first octet of 0) except IPv4 address 0.0.0.0 is treated as an interface index. An interface index is a 24-bit number, and the 0.0.0.0/8 IPv4 address block is not used (this range is reserved). The interface index can be used to specify the default interface for sending traffic for IPv4. The <a href="#">GetAdaptersAddresses</a> function can be used to obtain the interface index information. If <i>optval</i> is zero, the default interface for sending traffic is set to unspecified. When getting this option, the <i>optval</i> returns the current default interface index for sending IPv4 traffic in host byte order.</p>

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IP_USER_MTU	yes	yes	DWORD	<p>Gets or sets an upper bound on the IP layer MTU (in bytes) for the given socket. If the value is higher than the system's estimate of the path MTU (which you can retrieve on a connected socket by querying the <b>IP_MTU</b> socket option), then the option has no effect. If the value is lower, then outbound packets larger than this will be fragmented, or will fail to send, depending on the value of <b>IP_DONTFRAGMENT</b>. Default value is <b>IP_UNSPECIFIED_USER_MTU</b> (MAXULONG). For type-safety, you should use the <a href="#">WSAGetIPUserMtu</a> and <a href="#">WSASetIPUserMtu</a> functions instead of using the socket option directly.</p>
IP_WFP_REDIRECT_CONTEXT	yes	yes	WSACMSGHDR with control data	A datagram socket ancillary data type (cmsg_type) to indicate the redirect context for a UDP socket used by a user mode Windows Filtering Platform (WFP) redirect service.
IP_WFP_REDIRECT_RECORDS	yes	yes	WSACMSGHDR with control data	A datagram socket ancillary data type (cmsg_type) to indicate the redirect record for a UDP socket used by a user mode Windows Filtering Platform (WFP) redirect service.

## Windows support for IP\_PROTO options

OPTION	WINDOWS 10	WINDOWS 8	WINDOWS SERVER 2012	WINDOWS 7	WINDOWS SERVER 2008	WINDOWS VISTA
IP_ADD_IFLIST	Starting with Windows 10, version 1803					
IP_ADD_MEMBERSHIP	x	x	x	x	x	x
IP_ADD_SOURCE_MEMBERSHIP	x	x	x	x	x	x
IP_BLOCK_SOURCE	x	x	x	x	x	x
IP_DEL_IFLIST	Starting with Windows 10, version 1803					
IP_DONTFRAGMENT	x	x	x	x	x	x
IP_DROP_MEMBERSHIP	x	x	x	x	x	x
IP_DROP_SOURCE_MEMBERSHIP	x	x	x	x	x	x
IP_GET_IFLIST	Starting with Windows 10, version 1803					
IP_HDRINCL	x	x	x	x	x	x
IP_IFLIST	Starting with Windows 10, version 1803					
IP_MULTICAST_IF	x	x	x	x	x	x
IP_MULTICAST_LOOP	x	x	x	x	x	x
IP_MULTICAST_TTL	x	x	x	x	x	x
IP_OPTIONS	x	x	x	x	x	x
IP_ORIGINAL_ARRIVAL_IF	x	x	x	x		
IP_PKTINFO	x	x	x	x	x	x

OPTION	WINDOWS 10	WINDOWS 8	WINDOWS SERVER 2012	WINDOWS 7	WINDOWS SERVER 2008	WINDOWS VISTA
IP_RECEIVE_BROADCAST	x	x	x	x	x	x
IP_RECVIF	Starting with Windows 10, version 1703	x	x	x	x	x
IP_RECVTTL	x					
IP_TOS	x	x	x			
IP_TTL	x	x	x	x	x	x
IP_UNBLOCK_SOURCE	x	x	x	x	x	x
IP_UNICAST_IF	x	x	x	x	x	x
IP_WFP_REDIRRECT_CONTEXT	x	x	x			
IP_WFP_REDIRECT_RECORDS	x	x	x			

OPTION	WINDOWS SERVER 2003	WINDOWS XP
IP_ADD_IFLIST		
IP_ADD_MEMBERSHIP	x	x
IP_ADD_SOURCE_MEMBERSHIP	x	x
IP_BLOCK_SOURCE	x	x
IP_DEL_IFLIST		
IP_DONTFRAGMENT	x	x
IP_DROP_MEMBERSHIP	x	x
IP_DROP_SOURCE_MEMBERSHIP	x	x
IP_GET_IFLIST		
IP_HDRINCL	x	x

OPTION	WINDOWS SERVER 2003	WINDOWS XP
IP_IIFLIST		
IP_MULTICAST_IF	x	x
IP_MULTICAST_LOOP	x	x
IP_MULTICAST_TTL	x	x
IP_OPTIONS	x	x
IP_ORIGINAL_ARRIVAL_IF		
IP_PKTINFO	x	x
IP_RECEIVE_BROADCAST	x	x
IP_RECVIF		
IP_RECVTTL		
IP_TOS		
IP_TTL	x	x
IP_UNBLOCK_SOURCE	x	x
IP_UNICAST_IF		
IP_WFP_REDIRECT_CONTEXT		
IP_WFP_REDIRECT_RECORDS		

## Remarks

In the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and **IPPROTO\_IP** level is defined in the *Ws2def.h* header file which is automatically included in the *Winsock2.h* header file. Some of the **IPPROTO\_IP** socket options are defined in the *Ws2ipdef.h* header file which is automatically included by the *Ws2tcpip.h* header file. The remaining **IPPROTO\_IP** socket options are defined in the *Wsipv6ok.h* header file which is automatically included by the *Winsock2.h* header file. The *Ws2def.h*, *Ws2ipdef.h*, and *Wsipv6ok.h* header files should never be used directly.

In the Platform SDK released for Windows Server 2003 and Windows XP, the **IPPROTO\_IP** level is defined in the *Winsock2.h* header file. Some of the **IPPROTO\_IP** socket options are defined in the *Ws2tcpip.h* header file. The remaining **IPPROTO\_IP** socket options are defined in the *Wsipv6ok.h* header file which is automatically included by the *Winsock2.h* header file. The *Wsipv6ok.h* header file should never be used directly.

## Requirements

REQUIREMENT	VALUE
Header	Ws2def.h (include Winsock2.h); Ws2ipdef.h (include Ws2tcpip.h); Wsipv6ok.h (include Winsock2.h)

# IP\_PKTINFO socket option

3/5/2021 • 4 minutes to read • [Edit Online](#)

The IP\_PKTINFO socket option allows an application to enable or disable the return of packet information by the [LPFN\\_WSARECVMSG \(WSARecvMsg\)](#) function on an IPv4 socket..

To query the status of this socket option, call the [getsockopt](#) function. To set this option, call the [setsockopt](#) function with the following parameters.

## Socket option value

The constant that represents this socket option is 19.

## Syntax

```
int getsockopt(
    (SOCKET) s,      // descriptor identifying a socket
    (int) IPPROTO_IP, // level
    (int) IP_PKTINFO, // optname
    (char *) optval, // output buffer,
    (int) optlen,   // size of output buffer
);
```

```
int setsockopt(
    (SOCKET) s,      // descriptor identifying a socket
    (int) IPPROTO_IP, // level
    (int) IP_PKTINFO, // optname
    (char *) optval, // input buffer,
    (int) optlen,   // size of input buffer
);
```

## Parameters

*s* [in]

A descriptor identifying the socket.

*level* [in]

The level at which the option is defined. Use **IPPROTO\_IP** for this operation.

*optname* [in]

The socket option for which to get or set the value. Use **IP\_PKTINFO** for this operation.

*optval* [out]

A pointer to the buffer containing the value for the option to set. This parameter should point to buffer equal to or larger than the size of a **DWORD** value.

This value is treated as a boolean value with 0 used to indicate **FALSE** (disabled) and a nonzero value to indicate **TRUE** (enabled).

*optlen* [in, out]

A pointer to the size, in bytes, of the *optval*/buffer. This size must be equal to or larger than the size of a **DWORD** value.

## Return value

If the operation completes successfully, the [getsockopt](#) or [setsockopt](#) function returns zero.

If the operation fails, a value of **SOCKET\_ERROR** is returned and a specific error code can be retrieved by calling [WSAGetLastError](#).

ERROR CODE	MEANING
<a href="#">WSANOTINITIALISED</a>	A successful <a href="#">WSAStartup</a> call must occur before using this function.
<a href="#">WSAENETDOWN</a>	The network subsystem has failed.
<a href="#">WSAEFAULT</a>	One of the <i>optval</i> or the <i>optlen</i> parameters point to memory that is not in a valid part of the user address space. This error is also returned if the value pointed to by the <i>optlen</i> parameter is less than the size of a <b>DWORD</b> value.
<a href="#">WSAEINPROGRESS</a>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<a href="#">WSAEINVAL</a>	An invalid argument was supplied. This error is returned if the <i>level</i> /parameter is unknown or invalid. On Windows Vista and later, this error is also returned if the socket was in a transitional state.
<a href="#">WSAENOPROTOOPT</a>	The option is unknown or unsupported by the indicated protocol family. This error is returned if the <i>type</i> parameter for the socket descriptor passed in the <i>s</i> parameter was not <b>SOCK_DGRAM</b> or <b>SOCK_RAW</b> .
<a href="#">WSAENOTSOCK</a>	The descriptor is not a socket.

## Remarks

The [getsockopt](#) function called with the **IP\_PKTINFO** socket option allows an application to determine if packet information is to be returned by the [LPFN\\_WSARECVMSG \(WSARecvMsg\)](#) function for an IPv4 socket.

The [setsockopt](#) function called with the **IP\_PKTINFO** socket option allows an application to enable or disable the return of packet information by the [LPFN\\_WSARECVMSG \(WSARecvMsg\)](#) function. The **IP\_PKTINFO** option for a socket is disabled (set to **FALSE**) by default.

When this socket option is enabled on an IPv4 socket of type **SOCK\_DGRAM** or **SOCK\_RAW**, the [LPFN\\_WSARECVMSG \(WSARecvMsg\)](#) function returns packet information in the [WSAMSG](#) structure pointed to by the *lpMsg* parameter. One of the control data objects in the returned [WSAMSG](#) structure will contain an [in\\_pktinfo](#) structure used to store received packet address information.

For datagrams received by the [LPFN\\_WSARECVMSG \(WSARecvMsg\)](#) function over IPv4, the **Control**

member of the [WSAMSG](#) structure received will contain a [WSABUF](#) structure that contains a [WSACMSGHDR](#) structure. The [cmsg\\_level](#) member of this [WSACMSGHDR](#) structure would contain [IPPROTO\\_IP](#), the [cmsg\\_type](#) member of this structure would contain [IP\\_PKTINFO](#), and the [cmsg\\_data](#) member would contain an [in\\_pktinfo](#) structure used to store received IPv4 packet address information. The IPv4 address in the [in\\_pktinfo](#) structure is the IPv4 address from which the packet was received.

For a dual-stack datagram socket, if an application requires the [LPFN\\_WSARECVMMSG \(WSARecvMsg\)](#) function to return packet information in a [WSAMSG](#) structure for datagrams received over IPv4, then [IP\\_PKTINFO](#) socket option must be set to true on the socket. If only the [IPV6\\_PKTINFO](#) option is set to true on the socket, packet information will be provided for datagrams received over IPv6 but may not be provided for datagrams received over IPv4.

If an application tries to set the [IP\\_PKTINFO](#) socket option on a dual-stack datagram socket and IPv4 is disabled on the system, then the [setsockopt](#) function will fail and [WSAGetLastError](#) will return with an error of [WSAEINVAL](#). This same error is also returned by the [setsockopt](#) function as a result of other errors. If an application tries to set an [IPPROTO\\_IP](#) level socket option on a dual-stack socket and it fails with [WSAEINVAL](#), then the application should determine if IPv4 is disabled on the local computer. One method that can be used to detect if IPv4 is enabled or disabled is to call the [socket](#) function with the *af* parameter set to AF\_INET to try and create an IPv4 socket. If the [socket](#) function fails and [WSAGetLastError](#) returns an error of [WSAEAFNOSUPPORT](#), then it means IPv4 is not enabled. In this case, a [setsockopt](#) function failure when attempting to set the [IP\\_PKTINFO](#) socket option can be ignored by the application. Otherwise a failure when attempting to set the [IP\\_PKTINFO](#) socket option should be treated as an unexpected error.

Note that the *Ws2ipdef.h* header file is automatically included in *Ws2tcpip.h*, and should never be used directly.

## Requirements

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	Ws2ipdef.h (include Ws2tcpip.h)

## See also

[Dual-Stack Sockets](#)

[getsockopt](#)

[in\\_pktinfo](#)

[IPPROTO\\_IP Socket Options](#)

[IPV6\\_PKTINFO](#)

[setsockopt](#)

[socket](#)

[WSAMSG](#)

[LPFN\\_WSARECVMMSG \(WSARecvMsg\)](#)



# IPPROTO\_IPV6 socket options

6/3/2021 • 8 minutes to read • [Edit Online](#)

The following tables describe IPPROTO\_IPV6 socket options that apply to sockets created for the IPv6 address family (AF\_INET6). See the [getsockopt](#) and [setsockopt](#) function reference pages for more information on getting and setting socket options.

To enumerate protocols and discover supported properties for each installed protocol, use the [WSAEnumProtocols](#), [WSCEnumProtocols](#), or [WSCEnumProtocols32](#) function.

Some socket options require more explanation than these tables can convey; such options contain links to additional information.

## Options

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IP_ORIGINAL_ARRIVAL_IF	yes	yes	DWORD (boolean)	Indicates if the <a href="#">LPFN_WSARECVMSG (WSARecvMsg)</a> function should return optional control data containing the original arrival interface where the packet was received for datagram sockets. This option is used with IPv6 transition technologies (6to4, ISATAP, and Teredo tunnels, for example) that provide address assignment and host-to-host automatic tunneling for unicast IPv6 traffic when IPv6 hosts must traverse IP4 networks to reach other IPv6 networks. IPv6 packets are sent tunneled as IPv4 packets. This option allows the original IPv4 interface where the packet was received to be returned in the <a href="#">WSAMSG</a> structure.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IPV6_ADD_IFLIST		yes	DWORD (IF_INDEX)	Adds an interface index to the IFLIST associated with the IP_IFLIST option.
IPV6_ADD_MEMBERSHIP		yes	<a href="#">ipv6_mreq</a>	Join the socket to the supplied multicast group on the specified interface. This option is only valid on datagram and raw sockets (the socket type must be SOCK_DGRAM or SOCK_RAW).
IPV6_DEL_IFLIST		yes	DWORD (IF_INDEX)	Removes an interface index from the IFLIST associated with the IP_IFLIST option. Entries can be removed only by the application, so be aware that entries might go stale once an interface is removed.
IPV6_DROP_MEMBERSHIP		yes	<a href="#">ipv6_mreq</a>	Leave the supplied multicast group from the given interface. This option is only valid on datagram and raw sockets (the socket type must be SOCK_DGRAM or SOCK_RAW).
IPV6_GET_IFLIST	yes		DWORD[] (IF_INDEX[])	Gets the current IFLIST associated with the IP_IFLIST option. Returns error if IP_IFLIST is not enabled.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IPV6_HDRINCL	yes	yes	DWORD(boolean)	Indicates the application provides the IPv6 header on all outgoing data. If the <i>optval</i> /parameter is set to 1 on the call to <a href="#">setsockopt</a> , the option is enabled. If <i>optval</i> /is set to 0, the option is disabled. The default value is disabled. This option is only valid for datagram and raw sockets (the socket type must be SOCK_DGRAM or SOCK_RAW). A TCP/IP service provider that supports SOCK_RAW should also support IPV6_HDRINCL.
IPV6_HOPLIMIT	yes	yes	DWORD (boolean)	Indicates that hop (TTL) information should be returned in the <a href="#">LPFN_WSARECVM SG (WSARecvMsg)</a> function. If <i>optval</i> /is set to 1 on the call to <a href="#">setsockopt</a> , the option is enabled. If set to 0, the option is disabled. This option is only valid for datagram and raw sockets (the socket type must be SOCK_DGRAM or SOCK_RAW).
IPV6_IFLIST	yes	yes	DWORD (boolean)	Gets or sets the IP_IFLIST state of the socket. When this option is set to true, Datagram reception is restricted to interfaces that are in the IFLIST. Datagrams received on any other interfaces are ignored. IFLIST starts empty. Use <a href="#">IP_ADD_IFLIST</a> and <a href="#">IP_DEL_IFLIST</a> to edit the IFLIST.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IPV6_JOIN_GROUP		yes	<a href="#">ipv6_mreq</a>	Same as IPV6_ADD_MEMBERSHIP
IPV6_LEAVE_GROUP		yes	<a href="#">ipv6_mreq</a>	Same as IPV6_DROP_MEMBERSHIP
IPV6_MTU	yes		DWORD	Gets the system's estimate of the path MTU. Socket must be connected.
IPV6_MTU_DISCOVER	yes	yes	DWORD (PMTUD_STATE)	Gets or sets the path MTU discovery state for the socket. The default value is IP_PMTUDISC_NO_T_SET. For stream sockets, IP_PMTUDISC_NO_T_SET and IP_PMTUDISC_DO will perform path MTU discovery. IP_PMTUDISC_DONE and IP_PMTUDISC_PR_OBE will turn off path MTU discovery. For datagram sockets, if set to IP_PMTUDISC_DONE, attempts to send packets larger than path MTU will result in an error. If set to IP_PMTUDISC_DONE, packets will be fragmented according to interface MTU. If set to IP_PMTUDISC_PR_OBE, attempts to send packets larger than interface MTU will result in an error.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IPV6_MULTICAST_HOPS	yes	yes	DWORD	Gets or sets the TTL value associated with IPv6 multicast traffic on the socket. It is illegal to set the TTL to a value greater than 255. This option is only valid for datagram and raw sockets (the socket type must be SOCK_DGRAM or SOCK_RAW).
IPV6_MULTICAST_IF	yes	yes	DWORD	Gets or sets the outgoing interface for sending IPv6 multicast traffic. This option does not change the default interface for receiving IPv6 multicast traffic. This option is important for multihomed computers. The input value for setting this option is a 4-byte interface index of the desired outgoing interface in host byte order. The <a href="#">GetAdaptersAddresses</a> function can be used to obtain the interface index information. If <i>optval</i> is set to <b>NULL</b> on call to <a href="#">setsockopt</a> , the default IPv6 interface is used. If <i>optval</i> is zero , the default interface for receiving multicast is specified for sending multicast traffic. When getting this option, the <i>optval</i> returns the current default interface index for sending multicast IPv6 traffic in host byte order.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IPV6_MULTICAST_LOOP	yes	yes	DWORD (boolean)	Indicates multicast data sent on the socket will be echoed to the sockets receive buffer if it is also joined on the destination multicast group. If <i>optval</i> is set to 1 on the call to <a href="#">setsockopt</a> , the option is enabled. If set to 0, the option is disabled. This option is only valid for datagram and raw sockets (the socket type must be SOCK_DGRAM or SOCK_RAW).
IPV6_PKTINFO	yes	yes	DWORD (boolean)	Indicates that packet information should be returned by the <a href="#">LPFN_WSARECVM SG (WSARecvMsg)</a> function.
IPV6_PROTECTION_LEVEL	yes	yes	INT	Enables restriction of a socket to a specified scope, such as addresses with the same link local or site local prefix. Provides various restriction levels and default settings. See <a href="#">IPV6_PROTECTION_LEVEL</a> for more information.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IPV6_RECVIF	yes	yes	DWORD (boolean)	<p>Indicates whether the IP stack should populate the control buffer with details about which interface received a packet with a datagram socket. When this value is true, the <a href="#">LPFN_WSARECVM SG (WSARecvMsg)</a> function will return optional control data containing the interface where the packet was received for datagram sockets. This option allows the IPv6 interface where the packet was received to be returned in the <a href="#">WSAMSG</a> structure. This option is only valid for datagram and raw sockets (the socket type must be SOCK_DGRAM or SOCK_RAW).</p>

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IPV6_RECVTCLASS	yes	yes	DWORD (boolean)	<p>Indicates whether the IP stack should populate the control buffer with a message containing the Traffic Class IPv6 header field on a received datagram. When this value is true, the <a href="#">LPFN_WSARECVM SG (WSARecvMsg)</a> function will return optional control data containing the Traffic Class IPv6 header field value of the received datagram. This option allows the Traffic Class IPv6 header field of the received datagram to be returned in the <a href="#">WSAMSG</a> structure. The returned message type will be IPV6_TCLASS. All DSCP and ECN bits of the Traffic Class field will be returned. This option is only valid on datagram sockets (the socket type must be SOCK_DGRAM).</p>
IPV6_UNICAST_HOPS	yes	yes	DWORD	Gets or sets the current TTL value associated with IPv6 socket for unicast traffic. It is illegal to set the TTL to a value greater than 255.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IPV6_UNICAST_IF	yes	yes	DWORD (IF_INDEX)	<p>Gets or sets the outgoing interface for sending IPv6 traffic. This option does not change the default interface for receiving IPv6 traffic. This option is important for multihomed computers. The input value for setting this option is a 4-byte interface index of the desired outgoing interface in host byte order. The <a href="#">GetAdaptersAddresses</a> function can be used to obtain the interface index information. If <i>optval</i> is zero, the default interface for sending IPv6 traffic is set to unspecified. When getting this option, the <i>optval</i> returns the current default interface index for sending IPv6 traffic in host byte order.</p>

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IPV6_USER_MTU	yes	yes	DWORD	<p>Gets or sets an upper bound on the IP layer MTU (in bytes) for the given socket. If the value is higher than the system's estimate of the path MTU (which you can retrieve on a connected socket by querying the <b>IPV6_MTU</b> socket option), then the option has no effect. If the value is lower, then outbound packets larger than this will be fragmented, or will fail to send, depending on the value of <b>IPV6_DONTFRAG</b>. Default value is <b>IP_UNSPECIFIED_USER_MTU</b> (MAXULONG). For type-safety, you should use the <a href="#">WSAGetIPUserMtu</a> and <a href="#">WSASetIPUserMtu</a> functions instead of using the socket option directly.</p>

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IPV6_V6ONLY	yes	yes	DWORD (boolean)	<p>Indicates if a socket created for the AF_INET6 address family is restricted to IPv6 communications only. Sockets created for the AF_INET6 address family may be used for both IPv6 and IPv4 communications. Some applications may want to restrict their use of a socket created for the AF_INET6 address family to IPv6 communications only. When this value is nonzero (the default on Windows), a socket created for the AF_INET6 address family can be used to send and receive IPv6 packets only. When this value is zero, a socket created for the AF_INET6 address family can be used to send and receive packets to and from an IPv6 address or an IPv4 address. Note that the ability to interact with an IPv4 address requires the use of IPv4 mapped addresses. This socket option is supported on Windows Vista or later.</p>

## Windows support for IPPROTO\_IPV6 socket options

OPTION	WINDOWS 8	WINDOWS SERVER 2012	WINDOWS 7	WINDOWS SERVER 2008	WINDOWS VISTA
IP_ORIGINAL_AR RIVAL_IF	x	x	x		
IPV6_ADD_IFLIST	Starting with Windows 10, version 1803				

OPTION	WINDOWS 8	WINDOWS SERVER 2012	WINDOWS 7	WINDOWS SERVER 2008	WINDOWS VISTA
IPV6_ADD_MEMBERSHIP	x	x	x	x	x
IPV6_DEL_IFLIST	Starting with Windows 10, version 1803				
IPV6_DROP_MEMBERSHIP	x	x	x	x	x
IPV6_GET_IFLIST	Starting with Windows 10, version 1803				
IPV6_HDRINCL	x	x	x	x	x
IPV6_HOPLIMIT	x	x	x	x	x
IPV6_IFLIST	Starting with Windows 10, version 1803				
IPV6_JOIN_GROUP	x	x	x	x	x
IPV6_LEAVE_GROUP	x	x	x	x	x
IPV6_MULTICAST_HOPS	x	x	x	x	x
IPV6_MULTICAST_IF	x	x	x	x	x
IPV6_MULTICAST_LOOP	x	x	x	x	x
IPV6_PKTINFO	x	x	x	x	x
IPV6_PROTECTION_LEVEL	x	x	x	x	x
IPV6_RECVIF	x	x	x	x	x
IPV6_UNICAST_HOPS	x	x	x	x	x
IPV6_UNICAST_IF	x	x	x	x	x
IPV6_V6ONLY	x	x	x	x	x

OPTION	WINDOWS SERVER 2003	WINDOWS XP
IP_ORIGINAL_ARRIVAL_IF		
IPV6_ADD_IFLIST		
IPV6_ADD_MEMBERSHIP	x	x
IPV6_DEL_IFLIST		
IPV6_DROP_MEMBERSHIP	x	x
IPV6_GET_IFLIST		
IPV6_HDRINCL x	x	
IPV6_HOPLIMIT x	x	
IPV6_IFLIST		
IPV6_JOIN_GROUP	x	x
IPV6_LEAVE_GROUP	x	x
IPV6_MULTICAST_HOPS	x	x
IPV6_MULTICAST_IF	x	x
IPV6_MULTICAST_LOOP	x	x
IPV6_PKTINFO	x	x
IPV6_PROTECTION_LEVEL	x	x
IPV6_RECVIF		
IPV6_UNICAST_HOPS	x	x
IPV6_UNICAST_IF		
IPV6_V6ONLY		

## Remarks

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and **IPPROTO\_IPV6** level is defined in the *Ws2def.h* header file which is automatically included in the *Winsock2.h* header file. The **IPPROTO\_IPV6** socket options are defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The *Ws2def.h* and *Ws2ipdef.h* header files should never be used directly.

The IP\_ORIGINAL\_ARRIVAL\_IF socket option is supported on Windows Server 2008 R2 as well as on Windows 7.

## Requirements

Requirement	Value
Header	Ws2def.h (include Winsock2.h); Winsock2.h on Windows Server 2003 and Windows XP

# IPV6\_PKTINFO socket option

3/5/2021 • 3 minutes to read • [Edit Online](#)

The IPV6\_PKTINFO socket option allows an application to enable or disable the return of packet information by the [LPFN\\_WSARECVMMSG \(WSARecvMsg\)](#) function on an IPv6 socket..

To query the status of this socket option, call the [getsockopt](#) function. To set this option, call the [setsockopt](#) function with the following parameters.

## Socket option value

The constant that represents this socket option is 19.

## Syntax

```
int getsockopt(
    (SOCKET) s,      // descriptor identifying a socket
    (int) IPPROTO_IPV6, // level
    (int) IPV6_PKTINFO, // optname
    (char *) optval, // output buffer,
    (int) optlen, // size of output buffer
);
```

```
int setsockopt(
    (SOCKET) s,      // descriptor identifying a socket
    (int) IPPROTO_IPV6, // level
    (int) IPV6_PKTINFO, // optname
    (char *) optval, // input buffer,
    (int) optlen, // size of input buffer
);
```

## Parameters

*s* [in]

A descriptor identifying the socket.

*level* [in]

The level at which the option is defined. Use **IPPROTO\_IPV6** for this operation.

*optname* [in]

The socket option for which to get or set the value. Use **IPV6\_PKTINFO** for this operation.

*optval* [out]

A pointer to the buffer containing the value for the option to set. This parameter should point to buffer equal to or larger than the size of a **DWORD** value.

This value is treated as a boolean value with 0 used to indicate **FALSE** (disabled) and a nonzero value to indicate **TRUE** (enabled).

*optlen* [in, out]

A pointer to the size, in bytes, of the *optval*/buffer. This size must be equal to or larger than the size of a **DWORD** value.

## Return value

If the operation completes successfully, the [getsockopt](#) or [setsockopt](#) function returns zero.

If the operation fails, a value of **SOCKET\_ERROR** is returned and a specific error code can be retrieved by calling [WSAGetLastError](#).

ERROR CODE	MEANING
<a href="#">WSANOTINITIALISED</a>	A successful <a href="#">WSAStartup</a> call must occur before using this function.
<a href="#">WSAENETDOWN</a>	The network subsystem has failed.
<a href="#">WSAEFAULT</a>	One of the <i>optval</i> or the <i>optlen</i> parameters point to memory that is not in a valid part of the user address space. This error is also returned if the value pointed to by the <i>optlen</i> parameter is less than the size of a <b>DWORD</b> value.
<a href="#">WSAEINPROGRESS</a>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<a href="#">WSAEINVAL</a>	An invalid argument was supplied. This error is returned if the <i>level</i> /parameter is unknown or invalid. On Windows Vista and later, this error is also returned if the socket was in a transitional state.
<a href="#">WSAENOPROTOOPT</a>	The option is unknown or unsupported by the indicated protocol family. This error is returned if the <i>type</i> parameter for the socket descriptor passed in the <i>s</i> parameter was not <b>SOCK_DGRAM</b> or <b>SOCK_RAW</b> .
<a href="#">WSAENOTSOCK</a>	The descriptor is not a socket.

## Remarks

The [getsockopt](#) function called with the **IPV6\_PKTINFO** socket option allows an application to determine if packet information is to be returned by the [LPFN\\_WSARECVMSG \(WSARecvMsg\)](#) function for an IPv6 socket.

The [setsockopt](#) function called with the **IPV6\_PKTINFO** socket option allows an application to enable or disable the return of packet information by the [LPFN\\_WSARECVMSG \(WSARecvMsg\)](#) function. The **IPV6\_PKTINFO** option for a socket is disabled (set to **FALSE**) by default.

When this socket option is enabled on an IPv6 socket of type **SOCK\_DGRAM** or **SOCK\_RAW**, the [LPFN\\_WSARECVMSG \(WSARecvMsg\)](#) function returns packet information in the [WSAMSG](#) structure pointed to by the *lpMsg* parameter. One of the control data objects in the returned [WSAMSG](#) structure will contain an [in6\\_pktinfo](#) structure used to store received packet address information.

For datagrams received by the [LPFN\\_WSARECVMSG \(WSARecvMsg\)](#) function over IPv6, the **Control**

member of the **WSAMSG** structure received will contain a **WSABUF** structure that contains a **WSACMSGHDR** structure. The **cmsg\_level** member of this **WSACMSGHDR** structure would contain **IPPROTO\_IPV6**, the **cmsg\_type** member of this structure would contain **IPV6\_PKTINFO**, and the **cmsg\_data** member would contain an **in6\_pktinfo** structure used to store received IPv6 packet address information. The IPv6 address in the **in6\_pktinfo** structure is the IPv6 address from which the packet was received.

For a dual-stack datagram socket, if an application requires the **LPFN\_WSARECVMMSG (WSARecvMsg)** function to return packet information in a **WSAMSG** structure for datagrams received over IPv4, then **IP\_PKTINFO** socket option must be set to true on the socket. If only the **IPV6\_PKTINFO** option is set to true on the socket, packet information will be provided for datagrams received over IPv6 but may not be provided for datagrams received over IPv4.

Note that the *Ws2ipdef.h* header file is automatically included in *Ws2tcpip.h*, and should never be used directly.

## Requirements

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	Ws2ipdef.h (include Ws2tcpip.h)

## See also

[Dual-Stack Sockets](#)

[getsockopt](#)

[in6\\_pktinfo](#)

[IP\\_PKTINFO](#)

[IPPROTO\\_IPV6 Socket Options](#)

[setsockopt](#)

[socket](#)

[WSAMSG](#)

[LPFN\\_WSARECVMMSG \(WSARecvMsg\)](#)

# IPV6\_PROTECTION\_LEVEL

3/5/2021 • 4 minutes to read • [Edit Online](#)

The IPV6\_PROTECTION\_LEVEL socket option enables developers to place access restrictions on IPv6 sockets. Such restrictions enable an application running on a private LAN to simply and robustly harden itself against external attacks. The IPV6\_PROTECTION\_LEVEL socket option widens or narrows the scope of a listening socket, enabling unrestricted access from public and private users when appropriate, or restricting access only to the same site, as required.

IPV6\_PROTECTION\_LEVEL currently has three defined protection levels.

PROTECTION LEVEL	DESCRIPTION
PROTECTION_LEVEL_UNRESTRICTED	Used by applications designed to operate across the Internet, including applications taking advantage of IPv6 NAT traversal capabilities built into Windows (Teredo, for example). These applications may bypass IPv4 firewalls, so applications must be hardened against Internet attacks directed at the opened port.
PROTECTION_LEVEL_EDGERESTRICTED	Used by applications designed to operate across the Internet. This setting does not allow NAT traversal using the Windows Teredo implementation. These applications may bypass IPv4 firewalls, so applications must be hardened against Internet attacks directed at the opened port.
PROTECTION_LEVEL_RESTRICTED	Used by intranet applications that do not implement Internet scenarios. These applications are generally not tested or hardened against Internet-style attacks. This setting will limit the received traffic to link-local only.

The following code example provides the defined values for each:

```
#define PROTECTION_LEVEL_UNRESTRICTED 10 /* for peer-to-peer apps */
#define PROTECTION_LEVEL_EDGERESTRICTED 20 /* Same as unrestricted, except for Teredo */
#define PROTECTION_LEVEL_RESTRICTED 30 /* for Intranet apps */
```

These values are mutually exclusive, and cannot be combined in a single [setsockopt](#) function call. Other values for this socket option are reserved. These protection levels apply only to incoming connections. Setting this socket option has no effect on outbound packets or connections.

On Windows 7 and Windows Server 2008 R2, the default value for IPV6\_PROTECTION\_LEVEL is unspecified and PROTECTION\_LEVEL\_DEFAULT is defined to -1, an illegal value for IPV6\_PROTECTION\_LEVEL.

On Windows Vista and Windows Server 2008, the default value for IPV6\_PROTECTION\_LEVEL is PROTECTION\_LEVEL\_UNRESTRICTED and PROTECTION\_LEVEL\_DEFAULT is defined to -1, an illegal value for IPV6\_PROTECTION\_LEVEL.

On Windows Server 2003 and Windows XP, the default value for IPV6\_PROTECTION\_LEVEL is PROTECTION\_LEVEL\_EDGERESTRICTED and PROTECTION\_LEVEL\_DEFAULT is defined to be PROTECTION\_LEVEL\_EDGERESTRICTED.

**NOTE**

The IPV6\_PROTECTION\_LEVEL socket option should be set before the socket is bound. Otherwise, packets received between **bind** and **setsockopt** calls will conform to PROTECTION\_LEVEL\_EDGERESTRICTED, and may be delivered to the application.

The following table describes the effect of applying each protection level to a listening socket.

Protection level

Incoming traffic permitted

Same site

External

NAT traversal (Teredo)

PROTECTION\_LEVEL\_RESTRICTED

Yes

No

No

PROTECTION\_LEVEL\_EDGERESTRICTED

Yes

Yes

No

PROTECTION\_LEVEL\_UNRESTRICTED

Yes

Yes

Yes

In the table above, the **Same site** column is a combination of the following:

- Link local addresses
- Site Local addresses
- Global addresses known to belong to the same site (matching the site prefix table)

On Windows 7 and Windows Server 2008 R2, the default value for IPV6\_PROTECTION\_LEVEL is unspecified. If there is no edge-traversal aware firewall software installed on the local computer (Windows firewall is disabled or some other firewall is installed that ignores Teredo traffic), you will receive Teredo traffic only if you set the IPV6\_PROTECTION\_LEVEL socket option to PROTECTION\_LEVEL\_UNRESTRICTED. However, Windows firewall or any edge-traversal aware firewall policy may ignore this option based on policy settings for the firewall. By setting this socket option to PROTECTION\_LEVEL\_UNRESTRICTED, the application is communicating its explicit intent to receive edge traversed traffic by the host firewall installed on the local machine. So if there is an edge-traversal aware host firewall installed, it will have the final decision on accepting a packet. By default, without any socket option set:

- o If Windows firewall is enabled (or another edge-traversal aware host firewall is installed) on the local computer, whatever it enforces will be observed. Typical edge-traversal aware host firewall will block Teredo traffic by default. Therefore applications will observe the default as if it was **PROTECTION\_LEVEL\_EDGERESTRICTED**.
- o If Windows firewall is not enabled and no other edge-traversal aware host firewall is installed on the local system, the default will be **PROTECTION\_LEVEL\_EDGERESTRICTED**.

On Windows Vista and Windows Server 2008, the default value for IPV6\_PROTECTION\_LEVEL is **PROTECTION\_LEVEL\_UNRESTRICTED**. But the effective value depends on whether Windows firewall is enabled. The Windows firewall is edge-traversal aware (Teredo aware), no matter what value is set for the IPV6\_PROTECTION\_LEVEL and ignores if IPV6\_PROTECTION\_LEVEL is **PROTECTION\_LEVEL\_UNRESTRICTED**. So the effective value depends on the firewall policy. When Windows firewall is disabled and no other edge-traversal aware firewall is installed on the local computer, the default value for IPV6\_PROTECTION\_LEVEL is **PROTECTION\_LEVEL\_UNRESTRICTED**.

On Windows Server 2003 and Windows XP, the default value for IPV6\_PROTECTION\_LEVEL is **PROTECTION\_LEVEL\_EDGERESTRICTED**. Unless you have set the IPV6\_PROTECTION\_LEVEL socket option to **PROTECTION\_LEVEL\_UNRESTRICTED**, you would not receive any Teredo traffic.

Depending on the IPV6\_PROTECTION\_LEVEL, an application that requires unsolicited traffic from the Internet may not be capable of receiving unsolicited traffic. However, these requirements are not necessary for receiving solicited traffic over the Windows Teredo interface. For more details on the interaction with Teredo, see [Receiving Solicited Traffic Over Teredo](#).

When incoming packets or connections are refused due to the set protection level, rejection is handled as if no application was listening on that socket.

#### NOTE

The IPV6\_PROTECTION\_LEVEL socket option does not necessarily place access restrictions on IPv6 sockets or restrict NAT traversal using some method other than Windows Teredo or even using another implementation of Teredo by another vendor.

## Related topics

[getsockopt](#)

[Receiving Solicited Traffic Over Teredo](#)

[setsockopt](#)

# IPPROTO\_RM Socket Options

3/22/2021 • 3 minutes to read • [Edit Online](#)

The following table describes **IPPROTO\_RM** socket options that apply to sockets created for the IPv4 address family (AF\_INET) with the *protocol* parameter to the [socket](#) function specified as reliable multicast (IPPROTO\_RM). See the [getsockopt](#) and [setsockopt](#) function reference pages for more information on getting and setting socket options.

To enumerate protocols and discover supported properties for each installed protocol, use the [WSAEnumProtocols](#), [WSCEnumProtocols](#), or [WSCEnumProtocols32](#) function.

**Windows XP:** Reliable Multicast Programming (PGM) is not supported.

Some socket options require more explanation than these tables can convey; such options contain links to additional pages.

\*\*IPPROTO\_RM Socket Options\*\*

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
RM_ADD_RECEIVE_IF		yes	ULONG	Receiver only. Adds an interface on which to listen (the default is the first local interface enumerated). The optval parameter specifies the network interface in network byte order to add. The value specified replaces the default interface on the first call for a given socket, and adds other interfaces on subsequent calls. To obtain INADDR_ANY behavior, each network interface must be added separately.
RM_DEL_RECEIVE_IF		yes	ULONG	Receiver only. Removes an interface added using RM_ADD_RECEIVE_IF. The optval parameter specifies the network interface in network byte order to delete.
RM_FLUSHCACHE		yes	N/A	Not implemented.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
RM_HIGH_SPEED_INTRANET_OPT	yes	yes	ULONG	Receiver only. Specifies whether a high bandwidth LAN (100Mbps+) connection is used.
RM_LATEJOIN	yes	yes	ULONG	Sender only. Percentage of window size allowed to be requested by late-joining receivers upon session acceptance. Maximum value is 75% (default is zero). Disable this setting by calling again with value set to zero.
RM_RATE_WINDOW_SIZE	yes	yes	<a href="#">RM_SEND_WINDOW</a>	Sender only. Sets the transmission rate limit, window advance time, and window size.
RM_RECEIVER_STATISTICS	yes		<a href="#">RM_RECEIVER_STATS</a>	Receiver only. Retrieves statistics for the receiving session.
RM_SEND_WINDOW_ADV_RATE	yes	yes	ULONG	Sender only. Specifies the incremental advance rate for the trailing edge send window (default is 15%). Maximum value is 50%.
RM_SENDER_STATISTICS	yes		<a href="#">RM_SENDER_STATS</a>	Sender only. Retrieves statistics for the sending session.
RM_SENDER_WINDOW_ADVANCE_METHOD	yes	yes	ULONG	Sender only. The optval parameter specifies the method used when advancing the trailing edge send window. The optval parameter can only be E_WINDOW_ADVANCE_BY_TIME (the default). Note that E_WINDOW_USE_AS_DATA_CACHE is not supported.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
RM_SET_MCAST_TTL		yes	ULONG	Sender only. Sets the maximum time to live (TTL) setting for multicast packets. Maximum and default value is 255.
RM_SET_MESSAGE_BOUNDARY		yes	ULONG	Sender only. Specifies size of the next message to be sent, in bytes. Meaningful only to message mode sockets (SOCK_RDM). Can be set while the session is in progress.
RM_SET_SEND_IF	yes	yes	ULONG	Sender only. Sets the sending interface IP address in network byte order.
RM_USE_FEC	yes	yes	RM_FEC_INFO	Sender only. Notifies sender to apply forward error correction techniques to send repair data. FEC has three modes: pro-active parity packets only, OnDemand parity packets only, or both. See <a href="#">RM_FEC_INFO</a> structure for more information.

\*\*Windows Support for IPPROTO\_RM options\*\*

OPTION	WINDOWS 7	WINDOWS SERVER 2008	WINDOWS VISTA	WINDOWS SERVER 2003	WINDOWS XP	WINDOWS 2000	WINDOWS NT4	WINDOWS 9X/ME
RM_ADD_RECEIVE_IF	x	x	x	x	x			
RM_DEL_RECEIVE_IF	x	x	x	x	x			
RM_FLUSHCACHE	x	x	x	x	x			

OPTION	WINDOW S 7	WINDOW S SERVER 2008	WINDOW S VISTA	WINDOW S SERVER 2003	WINDOW S XP	WINDOW S 2000	WINDOW S NT4	WINDOW S 9X/ME
RM_HIG H_SPEED _INTRAN ET_OPT	x	x	x	x	x			
RM_LATE JOIN	x	x	x	x	x			
RM_RATE _WINDO W_SIZE	x	x	x	x	x			
RM_RECE IVER_STA TISTICS	x	x	x	x	x			
RM_SEN D_WIND OW_ADV _RATE	x	x	x	x	x			
RM_SEN DER_STAT ISTICS	x	x	x	x	x			
RM_SEN DER_WIN DOW_AD VANCE_ METHOD	x	x	x	x	x			
RM_SET_ MCAST_T TL	x	x	x	x	x			
RM_SET_ MESSAGE _BOUND ARY	x	x	x	x	x			
RM_SET_ SEND_IF	x	x	x	x	x			
RM_USE_ FEC	x	x	x	x	x			

## Remarks

The IPPROTO\_RM socket options and the structures used by these socket options are defined in the *Wsrm.h* header file.

The IPPROTO\_RM or the IPPROTO\_PGM constant can be used to specify the *protocol* parameter to the

[socket](#) function to use the RM socket options. On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the **IPPROTO\_PGM** constant is defined in the *Ws2def.h* header file to the same value as the **IPPROTO\_RM** constant defined in the *Wsrm.h* header file.

## Requirements

Requirement	Value
Header	Wsrm.h

# IPPROTO\_TCP socket options

3/22/2021 • 4 minutes to read • [Edit Online](#)

The following table describes **IPPROTO\_TCP** socket options that apply to sockets created for the IPv4 and IPv6 address families (AF\_INET and AF\_INET6) with the *protocol* parameter to the [socket](#) function specified as TCP (IPPROTO\_TCP). See the [getsockopt](#) and [setsockopt](#) function reference pages for more information on getting and setting socket options.

To enumerate protocols and discover supported properties for each installed protocol, use the [WSAEnumProtocols](#), [WSCEnumProtocols](#), or [WSCEnumProtocols32](#) function.

## Options

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
TCP_BSDURGENT	yes	yes	DWORD (Boolean)	If TRUE, the service provider implements the Berkeley Software Distribution (BSD) style (default) for handling expedited data. This option is the inverse of the TCP_EXPEDITED_112 2 option. This option can be set on the connection only once. Once this option is set on, this option cannot be turned off. This option is not required to be implemented by service providers. The option is enabled (set to TRUE) by default.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
TCP_EXPEDITED_112_2	yes	yes	DWORD (Boolean)	If TRUE, the service provider implements the expedited data as specified in RFC-1222. Otherwise, the Berkeley Software Distribution (BSD) style (default) is used. This option can be set on the connection only once. Once this option is set on, this option cannot be turned off. This option is not required to be implemented by service providers.
TCP_FAIL_CONNECT_ON_ICMP_ERROR	yes	yes	DWORD (Boolean)	If TRUE, a connect API call will return upon reception of an ICMP error with value WSAEHOSTUNREACH. The source address of the error will then be available via the TCP_ICMP_ERROR_INFORMATION socket option. If FALSE, the socket behaves normally. The default is disabled (set to FALSE). For type-safety, you should use the <a href="#">WSAGetFailConnectOnIcmpError</a> and <a href="#">WSASetFailConnectOnIcmpError</a> functions instead of using the socket option directly.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
TCP_ICMP_ERROR_INFO	yes	no	<a href="#">ICMP_ERROR_INFO</a>	<p>Retrieves the info of an ICMP error received by the TCP socket during a failed connect call. Only valid on a TCP socket where <code>TCP_FAIL_CONNECT_ON_ICMP_ERROR</code> has previously been enabled, and <code>connect</code> has returned <code>WSAEHOSTUNREACH</code>. The query is non-blocking. If queried successfully and the returned optlen value is 0, then no ICMP error has been received since the last connect call. If an ICMP error was received, its info will be available until <code>connect</code> is called again. The info is returned as an <a href="#">ICMP_ERROR_INFO</a> structure. For type-safety, you should use the <a href="#">WSAGetIcmpErrorInfo</a> function instead of using the socket option directly.</p>
TCP_KEEPCNT	yes	yes	DWORD	<p>Gets or sets the number of TCP keep alive probes that will be sent before the connection is terminated. It is illegal to set <code>TCP_KEEPCNT</code> to a value greater than 255.</p>

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
TCP_MAXRT	yes	yes	DWORD	If this value is non-negative, it represents the desired connection timeout in seconds. If it is -1, it represents a request to disable connection timeout (i.e. the connection will retransmit forever). If the connection timeout is disabled, the retransmit timeout increases exponentially for each retransmission up to its maximum value of 60sec and then stays there.
TCP_NODELAY	yes	yes	DWORD (Boolean)	Enables or disables the Nagle algorithm for TCP sockets. This option is disabled (set to FALSE) by default.
TCP_TIMESTAMP	yes	yes	DWORD (Boolean)	Enables or disables RFC 1323 time stamps. Note that there is also a global configuration for timestamps (default is off), "Timestamps" in (set/get)-nettcpsetting. Setting this socket option overrides that global configuration setting.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
TCP_FASTOPEN	yes	yes	DWORD (Boolean)	<p>Enables or disables <a href="#">RFC 7413</a> TCP Fast Open, which enables you to start sending data during the three-way handshake phase of opening a connection. Note that to make use of fast opens, you should use <a href="#">ConnectEx</a> to make the initial connection, and specify data in that function's <i>lpSendBuffer</i> parameter to be transferred during the handshake process. Some of the data in <i>lpSendBuffer</i> will be transferred under the Fast Open protocol.</p>
TCP_KEEPIDLE	yes	yes	DWORD	<p>Gets or sets the number of seconds a TCP connection will remain idle before keepalive probes are sent to the remote.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <b>[!Note]</b>            This option is available starting with Windows 10, version 1709.         </div>
TCP_KEEPINTVL	yes	yes	DWORD	<p>Gets or sets the number of seconds a TCP connection will wait for a keepalive response before sending another keepalive probe.</p> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <b>[!Note]</b>            This option is available starting with Windows 10, version 1709.         </div>

## Windows support for IPPROTO\_TCP options

OPTION	WINDOWS 10	WINDOWS 7	WINDOWS SERVER 2008	WINDOWS VISTA
TCP_BSDURGENT	x	x	x	x
TCP_EXPEDITED_1122	x	x	x	x
TCP_KEEPCNT	Starting with Windows 10, version 1703			
TCP_MAXRT	x	x	x	x
TCP_NODELAY	x	x	x	x
TCP_TIMESTAMPS	x	x	x	x
TCP_FASTOPEN	Starting with Windows 10, version 1607			

	OPTION	WINDOWS SERVER 2003	WINDOWS XP	WINDOWS 2000	WINDOWS NT4	WINDOWS 9X/ME
TCP_BSDURGENT	x	x	x	x		
TCP_EXPEDITED_1122	x	x	x			
TCP_KEEPCNT						
TCP_MAXRT						
TCP_NODELAY	x	x	x	x		
TCP_TIMESTAMPS						
TCP_FASTOPEN						

## Remarks

In the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and **IPPROTO\_TCP** level is defined in the *Ws2def.h* header file which is automatically included in the *Winsock2.h* header file. The **IPPROTO\_TCP** socket options, with the exception of **TCP\_BSDURGENT**, are defined in the *Ws2ipdef.h* header file which is automatically included in the *Ws2tcpip.h* header file. The **TCP\_BSDURGENT** option for historic reasons is defined in the *Mswsock.h* header file. The *Ws2def.h* and *Ws2ipdef.h* header files should never be used directly.

## Requirements

Requirement	Value
Header	Ws2def.h (include Winsock2.h); Winsock2.h on Windows Server 2003, Windows XP and Windows 2000

# IPPROTO\_UDP socket options

3/22/2021 • 2 minutes to read • [Edit Online](#)

The following table describes IPPROTO\_UDP socket options that apply to sockets created for the IPv4 and IPv6 address families (AF\_INET and AF\_INET6) with the *protocol* parameter to the [socket](#) function specified as UDP (IPPROTO\_UDP). See the [getsockopt](#) and [setsockopt](#) function reference pages for more information on getting and setting socket options.

To enumerate protocols and discover supported properties for each installed protocol, use the [WSAEnumProtocols](#), [WSCEnumProtocols](#), or [WSCEnumProtocols32](#) function.

## Options

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
UDP_CHECKSUM_C OVERAGE (ws2tcpip.h)	yes	yes	DWORD (boolean)	When TRUE, UDP datagrams are sent with a checksum.
UDP_NOCHECKSUM (ws2tcpip.h)	yes	yes	DWORD (boolean)	When TRUE, UDP datagrams are sent with the checksum of zero. Required for service providers. If a service provider does not have a mechanism to disable UDP checksum calculation, it may simply store this option without taking any action. This option is not supported for IPv6.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
UDP_RECV_MAX_COALESCED_SIZE (ws2ipdef.h; include ws2tcpip.h)	yes	yes	DWORD	<p>When set to a non-zero value, multiple received datagrams may be coalesced into a single message buffer before being indicated to your application. The option value represents the maximum message size in bytes for coalesced messages that can be indicated to your application. Un-coalesced messages larger than the option value may still be indicated. The default value is 0 (no coalescing). Datagrams will only be coalesced if they originated from the same source address and port. All datagrams coalesced will be of the same size—except the last datagram, which may be smaller. If your application wants to retrieve the datagram sizes (except the last datagram, which may differ) that were coalesced, you must use a receive API that supports control information (such as <a href="#">LPFN_WSARECVM SG (WSARecvMsg)</a>). The size of all but the last message can be found in the <b>UDP_COALESCED_INFO</b> control message, which is of type DWORD. For type safety, your application should use the <a href="#">WSAGetUdpRecvMax CoalescedSize</a> and <a href="#">WSASetUdpRecvMax CoalescedSize</a> functions instead of the socket option directly.</p>

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
UDP_SEND_MSG_SIZE (ws2ipdef.h; include ws2tcpip.h)	yes	yes	DWORD	<p>When set to a non-zero value, buffers sent by your application are broken down into multiple messages by the networking stack. The option value represents the size of each broken-down message. The option value is represented in bytes. The last segment's size may be less than the value of the option. The default value is 0 (no segmentation). Your application should set a value that is lower than the MTU of the path to the destination(s) in order to avoid IP fragmentation. For type safety, your application should use the <a href="#">WSAGetUdpSendMessageSize</a> and <a href="#">WSASetUdpSendMessageSize</a> functions instead of the socket option directly.</p>

## Legacy Windows support for IPPROTO\_UDP options

UDP\_CHECKSUM\_COVERAGE is unavailable on Windows 2000 and on Windows NT4.

UDP\_CHECKSUM\_COVERAGE and UDP\_NOCHECKSUM are unavailable on Windows 9x/Me.

## Remarks

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and IPPROTO\_UDP level is defined in the *Ws2def.h* header file which is automatically included in the *Winsock2.h* header file. The IPPROTO\_UDP socket options are defined in the *Ws2tcpip.h* header file. The *Ws2def.h* header file should never be used directly.

## Requirements

REQUIREMENT	VALUE
Header	ws2ipdef.h (include ws2tcpip.h), and ws2tcpip.h Winsock2.h on Windows Server 2003, Windows XP, and Windows 2000

# NSPROTO\_IPX Socket Options

3/22/2021 • 4 minutes to read • [Edit Online](#)

The following tables describe **NSPROTO\_IPX** socket options that apply to sockets created for the IPX/SPX address family (AF\_IPX). See the [getsockopt](#) and [setsockopt](#) function reference pages for more information on getting and setting socket options.

To enumerate protocols and discover supported properties for each installed protocol, use the [WSAEnumProtocols](#), [WSCEnumProtocols](#), or [WSCEnumProtocols32](#) function.

**\*\*NSPROTO\IPX Socket Options\*\***

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IPX_ADDRESS	yes		IPX_ADDRESS_DATA	Returns information about the specific adapter that IPX is enabled on.
IPX_ADDRESS_NOTIFY	yes		IPX_ADDRESS_DATA	Asynchronously notifies when the status of an IPX adapter changes.
IPX_DSTTYPE	yes	yes	DWORD	Gets or sets the value of the datastream field in the SPX header to send packets with.
IPX_EXTENDED_ADDRESS		yes	DWORD (boolean)	Enables the extended addressing option on IPX packets.
IPX_FILTERPTYPE	yes	yes	DWORD	Gets or sets the current IPX receive filter packet type. Only IPX packets with a packet type equal to the value specified in the optval parameter will be returned. Packets with a packet type that does not match are discarded. This is only applicable to a datagram socket.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IPX_GETNETINFO	yes		IPX_NETNUM_DATA	Returns information regarding a specific IPX network number. The netnum member of the <b>IPX_NETNUM_DATA</b> structure must be set to the IPX network number to be returned.
IPX_GETNETINFO_NO RIP	yes		IPX_NETNUM_DATA	Returns information regarding a specific IPX network number without sending a RIP request. The netnum member of the <b>IPX_NETNUM_DATA</b> structure must be set to the IPX network number to be returned.
IPX_IMMEDIATESPXA CK		yes	DWORD (boolean)	If set to <b>TRUE</b> , do not delay sending ACKs on an SPX connection.
IPX_MAX_ADAPTER_NUM	yes		DWORD	Returns the number of IPX enabled adapters present.
IPX_MAXSIZE	yes		DWORD	Returns the maximum IPX datagram size in bytes that can be sent.
IPX_PTYPE	yes	yes	DWORD	Gets or sets the packet type. The value specified in the optval parameter will be set as the packet type on every IPX packet sent from this socket.
IPX_RECEIVE_BROADCAST		yes	DWORD (boolean)	If set to <b>TRUE</b> , receive broadcast IPX packets.
IPX_RECVHDR		yes	DWORD (boolean)	If set to <b>TRUE</b> , receive IPX protocol headers with data.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IPX_RERIPNETNUMBE R	yes		IPX_NETNUM_DAT A	Returns information regarding a specified IPX network number using a new RIP request. The netnum member of the <b>IPX_NETNUM_DAT A</b> structure must be set to the IPX network number to be returned.
IPX_SPXGETCONNEC TIONSTATUS	yes		IPX_SPXCONNSTAT US_DATA	Returns information regarding a connected SPX socket statistics.
IPX_STOPFILTERPTYP E		yes	DWORD	Removes the filter and stops filtering on packet type specified in the optval parameter.

\*\*Windows Support for NSPROTO\\_IPX options\*\*

OPTION	WINDOWS VISTA AND LATER	WINDOWS SERVER 2003	WINDOWS XP	WINDOWS 2000	WINDOWS NT4	WINDOWS 9X/ME
IPX_ADDRESS		X	X	X	X	X
IPX_ADDRESS _NOTIFY		X	X	X	X	X
IPX_DSTYPE		X	X	X	X	X
IPX_EXTENDE D_ADDRESS		X	X	X	X	X
IPX_FILTERPT YPE		X	X	X	X	X
IPX_GETNETI NFO		X	X	X	X	X
IPX_GETNETI NFO_NORIP		X	X	X	X	X
IPX_IMMEDIA TESPXACK		X	X	X	X	X
IPX_MAX_A APTER_NUM		X	X	X	X	X

OPTION	WINDOWS VISTA AND LATER	WINDOWS SERVER 2003	WINDOWS XP	WINDOWS 2000	WINDOWS NT4	WINDOWS 9X/ME
IPX_MAXSIZE		x	x	x	x	x
IPX_PTYPE		x	x	x	x	x
IPX_RECEIVE_BROADCAST		x	x	x	x	x
IPX_RECVHDR		x	x	x	x	x
IPX_RERIPNETNUMBER		x	x	x	x	x
IPX_SPXGETCONNECTIONSTATUS		x	x	x	x	x
IPX_STOPFILTERTYPE		x	x	x	x	x

The following NSPROTO\_IPX socket options were defined in Windows Sockets 2 Protocol-Specific Annex, but are not implemented by the Windows IPX/SPX protocol.

/level= NSPROTO\_IPX

OPTION	TYPE	DEFAULT	MEANING
IPX_CHECKSUM	Bool	off	When set, IPX performs a checksum on outgoing packets and verifies the checksum of incoming packets.
IPX_TXPKTSIZE	int	Media size to a maximum of 1466	Sets the maximum send datagram size. This size does not include the IPX header or any media headers that may also be used. May be increased to media size.
IPX_RXPKTSIZE	int	Media size to a maximum of 1466	Sets the maximum receive datagram size. This size does not include the IPX header or any media headers that may also be used. May be increased to media size.
IPX_TXMEDIASIZE	int	Primary board	Returns the send media size that sets an upper bound for datagram size.

OPTION	TYPE	DEFAULT	MEANING
IPX_RXMEDIASIZE	int	Primary board	Returns the receive media size that sets an upper bound for datagram size.
IPX_PRIMARY	Bool	Primary	Restricts traffic to the primary network board.

The following **NSPROTO\_SPX** socket options were defined in Windows Sockets 2 Protocol-Specific Annex, but are not implemented on Windows by the Windows IPX/SPX protocol.

*/level* = **NSPROTO\_SPX**

OPTION	TYPE	DEFAULT	MEANING
SPX_CHECKSUM	Bool	off	When set, IPX performs a checksum on outgoing packets and verifies the checksum of incoming packets. Not supported on all platforms.
SPX_TXPKTSIZE	int	Media size to a maximum of 1466	Sets the maximum send datagram size. This size does not include the SPX header or any media headers that may also be used. May be increased to media size.
SPX_RXPKTSIZE	int	Media size to a maximum of 1466	Sets the maximum receive datagram size. This size does not include the SPX header or any media headers that may also be used. May be increased to media size.
SPX_TXMEDIASIZE	int	Primary board	Returns the send media size minus SPX and media headers. This sets an upper bound for message segmentation packet size.
SPX_RXMEDIASIZE	int	Primary board	Returns the receive media size minus SPX and media headers. This sets an upper bound for receive packet size.
SPX_RAWSPX	Bool	off	When set, the IPX/SPX protocol header is passed with the data.

## Remarks

The **NSPROTO\_IPX** socket options and the structures used by these socket options are defined in the *Wsnwlink.h* header file.

## Requirements

REQUIREMENT	VALUE
Header	Wsnwlink.h

# SOL\_APPLETALK Socket Options

3/22/2021 • 2 minutes to read • [Edit Online](#)

The following table describes **SOL\_APPLETALK** socket options that apply to sockets created for the AppleTalk address family (AF\_APPLETALK). See the [getsockopt](#) and [setsockopt](#) function reference pages for more information on getting and setting socket options.

To enumerate protocols and discover supported properties for each installed protocol, use the [WSAEnumProtocols](#), [WSCEnumProtocols](#), or [WSCEnumProtocols32](#) function.

**\*\*SOL\_APPLETALK Socket Options\*\***

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
SO_CONFIRM_NAME	yes		WSH_NBP_TUPLE	Confirms that a given AppleTalk name is bound to the given address.
SO_DEREGISTER_NAME		yes	WSH_REGISTER_NAME	Deregisters the name from the network.
SO_GETLOCALZONES	yes		WSH_LOOKUP_ZONES	Returns a list of zone names known to the given adapter name.
SO_GETMYZONE	yes		char []	Returns the default zone on the network.
SO_GETNETINFO	yes		WSH_LOOKUP_NETDEF_ON_ADAPTER	Returns the seeded values for the network as well as the default zone. The <i>optlen</i> parameter must be at least one byte larger than the size of the <b>WSH_LOOKUP_NETDEF_ON_ADAPTER</b> structure.
SO_GETZONELIST	yes		WSH_LOOKUP_ZONES	Returns zone names from the Internet zone list. The <i>optlen</i> parameter must be at least one byte larger than the size of the <b>WSH_LOOKUP_ZONES</b> structure.
SO_LOOKUP_MYZONE	yes			Same as SO_GETMYZONE

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
SO_LOOKUP_NAME	yes		WSH_LOOKUP_NAME	Looks up a specified NBP name and returns the matching tuples of name and NBP information. The <i>optlen</i> parameter must be at least one byte larger than the size of the WSH_LOOKUP_NAME structure.
SO_LOOKUP_NETDEF_ON_ADAPTER	yes		WSH_LOOKUP_NETDEF_ON_ADAPTER	Same as SO_GETNETINFO.
SO_LOOKUP_ZONES	yes		WSH_LOOKUP_ZONES	Same as SO_GETZONELIST.
SO_LOOKUP_ZONES_ON_ADAPTER	yes		WSH_LOOKUP_ZONES	Same as SO_GETLOCALZONES.
SO_PAP_GET_SERVER_STATUS	yes		WSH_PAP_GET_SERVER_STATUS	Returns the PAP status from a given server
SO_PAP_PRIME_READ		yes	char []	This call primes a read on a PAP connection so the sender can start sending the data
SO_PAP_SET_SERVER_STATUS		yes	char []	Sets the status to be sent if another client requests the status
SO_REGISTER_NAME		yes	WSH_REGISTER_NAME	Registers the given name on the AppleTalk network
SO_REMOVE_NAME		yes	WSH_REGISTER_NAME	Same as SO_DEREGISTER_NAME

\*\*Windows Support for SOL\_APPLETALK options\*\*

OPTION	WINDOWS VISTA AND LATER	WINDOWS SERVER 2003	WINDOWS XP	WINDOWS 2000	WINDOWS NT4	WINDOWS 9X/ME
SO_CONFIRM_NAME		X	X	X	X	

OPTION	WINDOWS VISTA AND LATER	WINDOWS SERVER 2003	WINDOWS XP	WINDOWS 2000	WINDOWS NT4	WINDOWS 9X/ME
SO_DEREGISTER_NAME		x	x	x	x	
SO_GETLOCALZONES		x	x	x	x	
SO_GETMYZONEONE		x	x	x	x	
SO_GETNETINFO		x	x	x	x	
SO_GETZONELIST		x	x	x	x	
SO_LOOKUP_MYZONE		x	x	x	x	
SO_LOOKUP_NAME		x	x	x	x	
SO_LOOKUP_NETDEF_ON_ADAPTER		x	x	x	x	
SO_LOOKUP_ZONES		x	x	x	x	
SO_LOOKUP_ZONES_ON_ADAPTER		x	x	x	x	
SO_PAP_GET_SERVER_STATUS		x	x	x	x	
SO_PAP_PRIME_READ		x	x	x	x	
SO_PAP_SET_SERVER_STATUS		x	x	x	x	
SO_REGISTER_NAME		x	x	x	x	
SO_REMOVE_NAME		x	x	x	x	

The **SOL\_APPLETALK** options are only supported on the server versions of Windows 2000 and Windows NT 4.0.

## Remarks

The SOL\_APPLETALK socket options and the structures used by these socket options are defined in the *Atalkwsh.h* header file.

## Requirements

REQUIREMENT	VALUE
Header	Atalkwsh.h

# SOL\_IRLMP Socket Options

3/22/2021 • 2 minutes to read • [Edit Online](#)

The following table describes **SOL\_IRLMP** socket options that apply to sockets created for the Infrared Data Association (IrDA) address family (AF\_IRDA) and the InfraRed Link Management Protocol (IRLMP). See the [getsockopt](#) and [setsockopt](#) function reference pages for more information on getting and setting socket options.

To enumerate protocols and discover supported properties for each installed protocol, use the [WSAEnumProtocols](#), [WSCEnumProtocols](#), or [WSCEnumProtocols32](#) function.

\*\*SOL\IRLMP Socket Options\*\*

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IRLMP_DISCOVERY_MODE				
IRLMP_ENUMDEVICES	yes		DEVICELIST	Returns a list of IrDA device IDs for IR capable devices within range.
IRLMP_EXCLUSIVE_MODE			DWORD (boolean)	Sets socket to bypass TinyTP layer to directly communicate with IrLMP.
IRLMP_IAS_QUERY	yes		IAS_QUERY	Queries IAS on a given service and class name for its attributes.
IRLMP_IAS_SET		yes	IAS_SET	Sets an attribute value for a given class name and attribute into IAS.
IRLMP_IRLPT_MODE		yes	DWORD (boolean)	Enables communication with IR capable printers.
IRLMP_PARAMETERS				
IRLMP_SEND_PDU_LEN	yes		DWORD	Retrieves the maximum PDU length required to use IRLMP_9WIRE_MODE.
IRLMP_SHARP_MODE		yes		

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
IRLMP_TINYTP_MODE		yes		
IRLMP_9WIRE_MODE	yes	yes	DWORD (boolean)	Puts the IrDA socket into IrCOMM mode.

## \*\*Windows Support for SOL\\_IRLMP options\*\*

## Remarks

The **SOL\_IRLMP** socket options and the structures used by these socket options are defined in the *Af\_irda.h* header file.

## Requirements

REQUIREMENT	VALUE
Header	<i>Af_irda.h</i>

# SOL\_SOCKET Socket Options

5/13/2021 • 12 minutes to read • [Edit Online](#)

The following tables describe SOL\_SOCKET socket options. See the [getsockopt](#) and [setsockopt](#) function reference pages for more information on getting and setting socket options.

To enumerate protocols and discover supported properties for each installed protocol, use the [WSAEnumProtocols](#), [WSCEnumProtocols](#), or [WSCEnumProtocols32](#) function.

Some socket options require more explanation than these tables can convey; such options contain links to additional pages.

## NOTE

All SOL\_SOCKET socket options apply equally to IPv4 and IPv6 (except SO\_BROADCAST, since broadcast is not implemented in IPv6).

## SOL\_SOCKET Socket Options

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
PVD_CONFIG	yes	yes	char []	An opaque data structure object containing configuration information for the service provider. This option is implementation dependent.
SO_ACCEPTCONN	yes		DWORD (boolean)	Returns whether a socket is in listening mode. This option is only Valid for connection-oriented protocols.
SO_BROADCAST	yes	yes	DWORD (boolean)	Configure a socket for sending broadcast data. This option is only Valid for protocols that support broadcasting (IPX and UDP, for example).

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
SO_BSP_STATE	yes		CSADDR_INFO	Returns the local address, local port, remote address, remote port, socket type, and protocol used by a socket. See the <a href="#">SO_BSP_STATE</a> reference for more information.
SO_CONDITIONAL_ACCEPT	yes	yes	DWORD (boolean)	Indicates if incoming connections are to be accepted or rejected by the application, not by the protocol stack. See the <a href="#">SO_CONDITIONAL_ACCEPT</a> reference for more information.
SO_CONNDATA	yes	yes	char []	Additional data, not in the normal network data stream, that is sent with network requests to establish a connection. This option is used by legacy protocols such as DECNet, OSI TP4, and others. This option is not supported by the TCP/IP protocol in Windows.
SO_CONNDALEN		yes	DWORD	The length, in bytes, of additional data, not in the normal network data stream, that is sent with network requests to establish a connection. This option is used by legacy protocols such as DECNet, OSI TP4, and others. This option is not supported by the TCP/IP protocol in Windows.
SO_CONNECT_TIME	yes		DWORD	Returns the number of seconds a socket has been connected. This option is only valid for connection-oriented protocols.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
SO_CONNOPT	yes	yes	char []	Additional connect option data, not in the normal network data stream, that is sent with network requests to establish a connection. This option is used by legacy protocols such as DECNet, OSI TP4, and others. This option is not supported by the TCP/IP protocol in Windows.
SO_CONNOPTLEN		yes	DWORD	The length, in bytes, of connect option data, not in the normal network data stream, that is sent with network requests to establish a connection. This option is used by legacy protocols such as DECNet, OSI TP4, and others. This option is not supported by the TCP/IP protocol in Windows.
SO_DISCDATA	yes	yes	char []	Additional data, not in the normal network data stream, that is sent with network requests to disconnect a connection. This option is used by legacy protocols such as DECNet, OSI TP4, and others. This option is not supported by the TCP/IP protocol in Windows.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
SO_DISCDATALEN		yes	DWORD	The length, in bytes, of additional data, not in the normal network data stream, that is sent with network requests to disconnect a connection. This option is used by legacy protocols such as DECNet, OSI TP4, and others. This option is not supported by the TCP/IP protocol in Windows.
SO_DISCOPT	yes	yes	char []	Additional disconnect option data, not in the normal network data stream, that is sent with network requests to disconnect a connection. This option is used by legacy protocols such as DECNet, OSI TP4, and others. This option is not supported by the TCP/IP protocol in Windows.
SO_DISCOPTLEN		yes	DWORD	The length, in bytes, of additional disconnect option data, not in the normal network data stream, that is sent with network requests to disconnect a connection. This option is used by legacy protocols such as DECNet, OSI TP4, and others. This option is not supported by the TCP/IP protocol in Windows.
SO_DEBUG	yes	yes	DWORD (boolean)	Enable debug output. Microsoft providers currently do not output any debug information.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
SO_DONTLINGER	yes	yes	DWORD (boolean)	Indicates the state of the <code>I_onoff</code> member of the <code>linger</code> structure associated with a socket. If this member is nonzero, a socket remains open for a specified amount of time after a <code>closesocket</code> function call to enable queued data to be sent. This option is only valid for reliable, connection-oriented protocols.
SO_DONTROUTE	yes	yes	DWORD (boolean)	Indicates that outgoing data should be sent on whatever interface the socket is bound to and not a routed on some other interface. This option is only Valid for message-oriented protocols. Microsoft providers silently ignore this option and always consult the routing table to find the appropriate outgoing interface.
SO_ERROR	yes		DWORD	Returns the last error code on this socket. This per-socket error code is not always immediately set.
SO_EXCLUSIVEADDRUSE	yes	yes	DWORD (boolean)	Prevents any other socket from binding to the same address and port. This option must be set before calling the <code>bind</code> function. See the <code>SO_EXCLUSIVEADDRUSE</code> reference for more information.
SO_GROUP_ID	yes		unsigned int	This socket option is reserved and should not be used.
SO_GROUP_PRIORITY	yes	yes	int	This socket option is reserved and should not be used.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
<a href="#">SO_KEEPALIVE</a>	yes	yes	DWORD (boolean)	Enables keep-alive for a socket connection. Valid only for protocols that support the notion of keep-alive (connection-oriented protocols). For TCP, the default keep-alive timeout is 2 hours and the keep-alive interval is 1 second. The default number of keep-alive probes varies based on the version of Windows. See the <a href="#">SO_KEEPALIVE</a> reference for more information.
SO_LINGER	yes	yes	struct linger	Indicates the state of the <a href="#">linger</a> structure associated with a socket. If the <a href="#">L_onoff</a> member of the <a href="#">linger</a> structure is nonzero, a socket remains open for a specified amount of time after a <a href="#">closesocket</a> function call to enable queued data to be sent. The amount of time, in seconds, to remain open is specified in the <a href="#">L_linger</a> member of the <a href="#">linger</a> structure. This option is only valid for reliable, connection-oriented protocols.
SO_MAX_MSG_SIZE	yes		DWORD	Returns the maximum outbound message size for message-oriented sockets supported by the protocol. Has no meaning for stream-oriented sockets.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
SO_MAXDG	yes		DWORD	Returns the maximum size, in bytes, for outbound datagrams supported by the protocol. This socket option has no meaning for stream-oriented sockets.
SO_MAXPATHDG	yes		DWORD	Returns the maximum size, in bytes, for outbound datagrams supported by the protocol to a given destination address. This socket option has no meaning for stream-oriented sockets. Microsoft providers may silently treat this as SO_MAXDG.
SO_OOBINLINE	yes	yes	DWORD (boolean)	Indicates that out-of-bound data should be returned in-line with regular data. This option is only valid for connection-oriented protocols that support out-of-band data.
SO_OPENTYPE	yes	yes	DWORD	Once set, affects whether subsequent sockets that are created will be non-overlapped. The possible values for this option are SO_SYNCHRONOUS_ALERT and SO_SYNCHRONOUS_NONALERT. This option should not be used. Instead use the <a href="#">WSASocket</a> function and leave the WSA_FLAG_OVERLAPPED bit in the <i>dwFlags</i> parameter turned off.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
SO_PAUSE_ACCEPT	yes	yes	DWORD(boolean)	Use this option for listening sockets. When the option is set, the socket responds to all incoming connections with an RST rather than accepting them.
SO_PORT_SCALABILITY	yes	yes	DWORD (boolean)	Enables local port scalability for a socket by allowing port allocation to be maximized by allocating wildcard ports multiple times for different local address port pairs on a local machine. On platforms where both options are available, prefer SO_REUSE_UNICAST PORT instead of this option. See the <a href="#">SO_PORT_SCALABILITY</a> reference for more information.
SO_PROTOCOL_INFO	yes		<a href="#">WSAPROTOCOL_INF</a> O	This option is defined to the SO_PROTOCOL_INFO W socket option if the UNICODE macro is defined. If the UNICODE macro is not defined, then this option is defined to the SO_PROTOCOL_INFO A socket option.
SO_PROTOCOL_INFO A	yes		<a href="#">WSAPROTOCOL_INF</a> O A	Returns the <a href="#">WSAPROTOCOL_INF</a> O A structure for the given socket
SO_PROTOCOL_INFO W	yes		<a href="#">WSAPROTOCOL_INF</a> O W	Returns the <a href="#">WSAPROTOCOL_INF</a> O W structure for the given socket

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
SO_RANDOMIZE_PORT	yes	yes	DWORD(boolean)	This option should be set on an unbound socket. When SO_RANDOMIZE_PORT is set and an ephemeral port is selected on the socket, a random port number is bound. Auto-reuse ports (ports selected using SO_REUSE_UNICAST_PORT) also randomize the returned port, so if an application sets SO_REUSE_UNICAST_PORT and then attempts to set SO_RANDOMIZE_PORT, the second <a href="#">setsockopt</a> call fails.
SO_RCVBUF	yes	yes	DWORD	The total per-socket buffer space reserved for receives. This is unrelated to SO_MAX_MSG_SIZE and does not necessarily correspond to the size of the TCP receive window.
SO_RCVLOWAT	yes	yes	DWORD	A socket option from BSD UNIX included for backward compatibility. This option sets the minimum number of bytes to process for socket input operations. This option is not supported by the Windows TCP/IP provider. If this option is used on Windows Vista and later, the <a href="#">getsockopt</a> and <a href="#">setsockopt</a> functions fail with WSAEINVAL. On earlier versions of Windows, these functions fail with WSAENOPROTOOPT.

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
SO_RCVTIMEO	yes	yes	DWORD	<p>The timeout, in milliseconds, for blocking receive calls. The default for this option is zero, which indicates that a receive operation will not time out. If a blocking receive call times out, the connection is in an indeterminate state and should be closed. If the socket is created using the <a href="#">WSASocket</a> function, then the <i>dwFlags</i> parameter must have the WSA_FLAG_OVERLAPPED attribute set for the timeout to function properly. Otherwise the timeout never takes effect.</p>
SO_REUSEADDR	yes	yes	DWORD (boolean)	Allows a socket to bind to an address and port already in use. The SO_EXCLUSIVEADDRUSE option can prevent this.
SO_REUSE_UNICAST_PORT	yes	yes	DWORD (boolean)	<p>When set, allow ephemeral port reuse for Winsock API connection functions which require an explicit bind, such as <a href="#">ConnectEx</a>. Note that connection functions with an implicit bind (such as <a href="#">connect</a> without an explicit <a href="#">bind</a>) have this option set by default. Use this option instead of <a href="#">SO_PORT_SCALABILITY</a> on platforms where both are available.</p>

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
SO_REUSE_MULTICASTPORT	yes		DWORD	<p>When set on a socket, this option indicates that the socket will never be used to receive unicast packets, and consequently that its port can be shared with other multicast-only applications.</p> <p>Setting the value to 1 enables always sharing multicast traffic on the port.</p> <p>Setting the value to 0 (default) disables this behavior.</p>
SO_SNDBUF	yes	yes	DWORD	<p>The total per-socket buffer space reserved for sends. This is unrelated to SO_MAX_MSG_SIZE and does not necessarily correspond to the size of a TCP send window.</p>
SO_SNDLOWAT	yes	yes	DWORD	<p>A socket option from BSD UNIX included for backward compatibility. This option sets the minimum number of bytes to process for socket output operations. This option is not supported by the Windows TCP/IP provider. If this option is used on Windows Vista and later, the <a href="#">getsockopt</a> and <a href="#">setsockopt</a> functions fail with WSAEINVAL. On earlier versions of Windows, these functions fail with WSAENOPROTOOPT.</p>

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
SO_SNDTIMEO	yes	yes	DWORD	<p>The timeout, in milliseconds, for blocking send calls. The default for this option is zero, which indicates that a send operation will not time out. If a blocking send call times out, the connection is in an indeterminate state and should be closed. If the socket is created using the <a href="#">WSASocket</a> function, then the <i>dwFlags</i> parameter must have the WSA_FLAG_OVERLAPPED attribute set for the timeout to function properly. Otherwise the timeout never takes effect.</p>
SO_TYPE	yes		DWORD	Returns the socket type for the given socket (SOCK_STREAM or SOCK_DGRAM, for example).
SO_UPDATE_ACCEPT_CONTEXT		yes	DWORD (boolean)	<p>This option is used with the <a href="#">AcceptEx</a> function. This option updates the properties of the socket which are inherited from the listening socket. This option should be set if the <a href="#">getpeername</a>, <a href="#">getsockname</a>, <a href="#">getsockopt</a>, or <a href="#">setsockopt</a> functions are to be used on the accepted socket.</p>

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
SO_UPDATE_CONNECT_CONTEXT		yes	DWORD (boolean)	This option is used with the <a href="#">ConnectEx</a> , <a href="#">WSAConnectByList</a> , and <a href="#">WSAConnectByName</a> functions. This option updates the properties of the socket after the connection is established. This option should be set if the <a href="#">getpeername</a> , <a href="#">getsockname</a> , <a href="#">getsockopt</a> , <a href="#">setsockopt</a> , or <a href="#">shutdown</a> functions are to be used on the connected socket.
SO_USELOOPBACK	yes	yes	DWORD (boolean)	Use the local loopback address when sending data from this socket. This option should only be used when all data sent will also be received locally. This option is not supported by the Windows TCP/IP provider. If this option is used on Windows Vista and later, the <a href="#">getsockopt</a> and <a href="#">setsockopt</a> functions fail with WSAEINVAL. On earlier versions of Windows, these functions fail with WSAENOPROTOOPT.

## Windows Support for SOL\_SOCKET Options







OPTION	WINDO WS 10	WINDO WS 7	WINDO WS SERVER 2008	WINDO WS VISTA	WINDO WS SERVER 2003	WINDO WS XP	WINDO WS 2000	WINDO WS NT4	WINDO WS 9X/ME
SO_UPD ATE_CO NNECT_ CONTEX T	x	x	x	x	x	x			
SO_USE LOOPB ACK									

## Remarks

The **SOL\_SOCKET** socket options are defined in several Winsock header files:

- *Winsock2.h*
- *Mswsock.h*
- *Ws2def.h*

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and **SOL\_SOCKET** level is defined in the *Ws2def.h* header file which is automatically included in the *Winsock2.h* header file. Some of the **SOL\_SOCKET** socket options are defined in the *Winsock2.h* and *Mswsock.h* header files. The remaining **SOL\_SOCKET** socket options are defined in the *Ws2def.h* header file which is automatically included by the *Winsock2.h* header file. The *Ws2def.h* should never be used directly.

On the Platform Software Development Kit (SDK) released for Windows Server 2003 and Windows XP, the **SOL\_SOCKET** level is defined in the *Winsock2.h* header file. The **SOL\_SOCKET** socket options are defined in the *Winsock2.h* and *Mswsock.h* header files.

## Requirements

REQUIREMENT	VALUE
Header	Winsock2.h; Mswsock.h; Ws2def.h (include Winsock2.h)

# SO\_BSP\_STATE socket option

3/5/2021 • 3 minutes to read • [Edit Online](#)

The **SO\_BSP\_STATE** socket option returns the local address, local port, remote address, remote port, socket type, and protocol used by a socket.

To perform this operation, call the [getsockopt](#) function with the following parameters.

## Socket option value

The constant that represents this socket option is 0x1009.

## Syntax

```
int getsockopt(  
    (SOCKET) s,      // descriptor identifying a socket  
    (int) SOL_SOCKET, // level  
    (int) SO_BSP_STATE, // optname  
    (char *) optval,      // output buffer,  
    (int) *optlen,       // size of output buffer  
) ;
```

## Parameters

*s* [in]

A descriptor identifying the socket.

*level* [in]

The level at which the option is defined. Use **SOL\_SOCKET** for this operation.

*optname* [in]

The socket option for which the value is to be retrieved. Use **SO\_BSP\_STATE** for this operation.

*optval* [out]

A pointer to the buffer in which the value for the requested option is to be returned. This parameter should point to buffer equal to or larger than the size of a [CSADDR\\_INFO](#) structure.

*optlen* [in, out]

A pointer to the size, in bytes, of the *optval* buffer. This size must be equal to or larger than the size of a [CSADDR\\_INFO](#) structure.

## Return value

If the operation completes successfully, [getsockopt](#) returns zero.

If the operation fails, a value of **SOCKET\_ERROR** is returned and a specific error code can be retrieved by calling [WSAGetLastError](#).

ERROR CODE	MEANING
<b>WSANOTINITIALISED</b>	A successful <a href="#">WSAStartup</a> call must occur before using this function.
<b>WSAENETDOWN</b>	The network subsystem has failed.
<b>WSAEFAULT</b>	One of the <i>optval</i> or the <i>optlen</i> parameters point to memory that is not in a valid part of the user address space. This error is also returned if the value pointed to by the <i>optlen</i> parameter is less than the size of a <a href="#">CSADDR_INFO</a> structure.
<b>WSAEINPROGRESS</b>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<b>WSAEINVAL</b>	The <i>level</i> parameter is unknown or invalid.
<b>WSAENOPROTOOPT</b>	The option is unknown or unsupported by the indicated protocol family.
<b>WSAENOTSOCK</b>	The descriptor is not a socket.

## Remarks

The [getsockopt](#) function called with the **SO\_BSP\_STATE** socket option retrieves the local address, local port, remote address, remote port, socket type, and protocol used by a socket. The **SO\_BSP\_STATE** socket option works with IPv6 or IPv4 sockets (the **AF\_INET6** and **AF\_INET** address families).

If the [getsockopt](#) function is successful, the information is returned in a [CSADDR\\_INFO](#) structure in the buffer pointed to by the *optval* parameter. The integer pointed to by *optlen* should originally contain the size of this buffer; on return, it will be set to the length, in bytes, of the value returned in the *optval* parameter.

The **iSocketType** and **iProtocol** members in the returned [CSADDR\\_INFO](#) structure are filled in for the socket descriptor in the *s* parameter.

If the socket is in a connected or bound state, then the **LocalAddr** member of the returned [CSADDR\\_INFO](#) structure will be set to a [SOCKADDR](#) structure representing the local address and port. If the socket is in a connected state, then the **RemoteAddr** member of the returned [CSADDR\\_INFO](#) structure will be set to a [SOCKADDR](#) structure representing the remote address and port.

If the socket is not in a connected or bound state, then the **LocalAddr** member of the returned [CSADDR\\_INFO](#) structure is returned with a **NULL** pointer in the **IpSockaddr** member and the **iSockaddrLength** member set to zero. If the socket is not in a bound state, then the **RemoteAddr** member of the returned [CSADDR\\_INFO](#) structure is returned with a **NULL** pointer in the **IpSockaddr** member and the **iSockaddrLength** member set to zero.

If the [getsockopt](#) function fails, the *optval* and *optlen* parameters are left unchanged and the *optval* parameter does not point to a returned [CSADDR\\_INFO](#) structure.

Note that the *Ws2def.h* header file is automatically included in *Winsock2.h*, and should never be used directly.

## Requirements

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	Ws2def.h (include Winsock2.h)

## See also

[getsockopt](#)

[CSADDR\\_INFO](#)

[SOCKADDR](#)

# SO\_CONDITIONAL\_ACCEPT socket option

3/5/2021 • 4 minutes to read • [Edit Online](#)

The SO\_CONDITIONAL\_ACCEPT socket option is designed to allow an application to decide whether or not to accept an incoming connection on a listening socket.

## Socket option value

The constant that represents this socket option is 0x3002.

## Syntax

```
int setsockopt(  
    (SOCKET) s,           // descriptor identifying a socket  
    (int) SOL_SOCKET,     // level  
    (int) SO_CONDITIONAL_ACCEPT, // optname  
    (char *) optval,       // input buffer,  
    (int) optlen,          // size of input buffer  
);
```

## Parameters

*s* [in]

A descriptor identifying the socket.

*level* [in]

The level at which the option is defined. Use SOL\_SOCKET for this operation.

*optname* [in]

The socket option for which the value is to be set. Use SO\_CONDITIONAL\_ACCEPT for this operation.

*optval* [out]

A pointer to the buffer containing the value for the option to set. This parameter should point to buffer equal to or larger than the size of a DWORD value.

This value is treated as a boolean value with 0 used to indicate FALSE (disabled) and a nonzero value to indicate TRUE (enabled).

*optlen* [in, out]

A pointer to the size, in bytes, of the *optval*/buffer. This size must be equal to or larger than the size of a DWORD value.

## Return value

If the operation completes successfully, **setsockopt** returns zero.

If the operation fails, a value of SOCKET\_ERROR is returned and a specific error code can be retrieved by calling **WSAGetLastError**.

ERROR CODE	MEANING
<b>WSANOTINITIALISED</b>	A successful <a href="#">WSAStartup</a> call must occur before using this function.
<b>WSAENETDOWN</b>	The network subsystem has failed.
<b>WSAEFAULT</b>	One of the <i>optval</i> or the <i>optlen</i> parameters point to memory that is not in a valid part of the user address space. This error is also returned if the value pointed to by the <i>optlen</i> parameter is less than the size of a DWORD value.
<b>WSAEINPROGRESS</b>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<b>WSAEINVAL</b>	The <i>level</i> parameter is unknown or invalid. This error is also returned if the socket was already in a listening state.
<b>WSAENOPROTOOPT</b>	The option is unknown or unsupported by the indicated protocol family.
<b>WSAENOTSOCK</b>	The descriptor is not a socket.

## Remarks

The [setsockopt](#) function called with the **SO\_CONDITIONAL\_ACCEPT** socket option allows an application to decide whether or not to accept an incoming connection on a listening socket. The conditional accept option for a socket is disabled (set to **FALSE**) by default.

When this socket option is enabled, the TCP stack does not automatically accept connections. It waits for the application to indicate that it accepts the connection via the conditional accept callback registered with [WSAAccept](#) function. Once a connection request is received, Winsock invokes the conditional accept callback registered by the application. Only when the conditional accept callback returns **CF\_ACCEPT** will Winsock notify the transport to fully accept the connection.

When the **SO\_CONDITIONAL\_ACCEPT** socket option is set to enabled (set to **TRUE**), this has several side effects on the socket:

- This disables the TCP stack's built-in SYN attack protection defenses, since the application is now taking ownership of accepting connections.
- This effectively disables the listen backlog since connections aren't accepted on behalf of a listening socket. A connection is never fully accepted until the application fully accepts using the **CF\_ACCEPT** mechanism.
- This also means the application take care to always be in a state to readily handle the accept callbacks to accept the connection. If the application is busy doing other processing, then the client side may time out before the application actually accepts the connection. This leads to a poor client experience.

Generally, the main reason applications use conditional accept is to inspect the source IP address or port used by the connecting client and then decide to accept or reject the connection. However, applications are likely to perform better if the **SO\_CONDITIONAL\_ACCEPT** option is not enabled and the application allows the TCP stack

and the listen backlog to work as expected. Then when the application handles the accepted connection, if it is from a improper IP source address or port, the application can just close the socket. The security drawback to this behavior is that now the client has confirmation that the application is listening on an IP address and port since it forcefully closed the previously accepted connection. However, the drawbacks to enabling `SO_CONDITIONAL_ACCEPT` generally outweigh this limitation.

The [getsockopt](#) function called with the `SO_CONDITIONAL_ACCEPT` socket option allows an application to retrieve the current state of the conditional accept option, although this is feature not normally used. If an application needs to enable conditional accept on a socket, it justs calls the [setsockopt](#) function to enable the option.

Note that the `Ws2def.h` header file is automatically included in `Winsock2.h`, and should never be used directly.

## Requirements

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	<code>Ws2def.h</code> (include <code>Winsock2.h</code> )

## See also

[getsockopt](#)

[setsockopt](#)

[WSAAccept](#)

# SO\_EXCLUSIVEADDRUSE socket option

3/5/2021 • 6 minutes to read • [Edit Online](#)

The SO\_EXCLUSIVEADDRUSE socket option prevents other sockets from being forcibly bound to the same address and port.

## Syntax

The SO\_EXCLUSIVEADDRUSE option prevents other sockets from being forcibly bound to the same address and port, a practice enabled by the SO\_REUSEADDR socket option. Such reuse can be executed by malicious applications to disrupt the application. The SO\_EXCLUSIVEADDRUSE option is very useful to system services requiring high availability.

The following code example illustrates setting this option.

```
#ifndef UNICODE
#define UNICODE
#endif

#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <stdio.h>
#include <stdlib.h>           // Needed for _wtoi

#pragma comment(lib, "Ws2_32.lib")

int __cdecl wmain(int argc, wchar_t ** argv)
{
    WSADATA wsaData;
    int iResult = 0;
    int iError = 0;

    SOCKET s = INVALID_SOCKET;
    SOCKADDR_IN saLocal;
    int iOptval = 0;

    int iFamily = AF_UNSPEC;
    int iType = 0;
    int iProtocol = 0;

    int iPort = 0;

    // Validate the parameters
    if (argc != 5) {
        wprintf(L"usage: %ws <addressfamily> <type> <protocol> <port>\n", argv[0]);
        wprintf(L"    opens a socket for the specified family, type, & protocol\n");
        wprintf(L"    sets the SO_EXCLUSIVEADDRUSE socket option on the socket\n");
        wprintf(L"    then tries to bind the port specified on the command-line\n");
        wprintf(L"%ws example usage\n", argv[0]);
        wprintf(L"    %ws 0 2 17 5150\n", argv[0]);
        wprintf(L"    where AF_UNSPEC=0 SOCK_DGRAM=2 IPPROTO_UDP=17 PORT=5150\n",
               argv[0]);
        wprintf(L"    %ws 2 1 17 5150\n", argv[0]);
        wprintf(L"    where AF_INET=2 SOCK_STREAM=1 IPPROTO_TCP=6 PORT=5150\n", argv[0]);
        wprintf(L"    See the documentation for the socket function for other values\n");
    }
}
```

```

        return 1;
    }

    iFamily = _wtoi(argv[1]);
    iType = _wtoi(argv[2]);
    iProtocol = _wtoi(argv[3]);

    iPort = _wtoi(argv[4]);

    if (iFamily != AF_INET && iFamily != AF_INET6) {
        wprintf(L"Address family must be either AF_INET (2) or AF_INET6 (23)\n");
        return 1;
    }

    if (iPort <= 0 || iPort >= 65535) {
        wprintf(L"Port specified must be greater than 0 and less than 65535\n");
        return 1;
    }

    // Initialize Winsock
    iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (iResult != 0) {
        wprintf(L"WSAStartup failed with error: %d\n", iResult);
        return 1;
    }

    // Create the socket
    s = socket(iFamily, iType, iProtocol);
    if (s == INVALID_SOCKET) {
        wprintf(L"socket failed with error: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    // Set the exclusive address option
    iOptval = 1;
    iResult = setsockopt(s, SOL_SOCKET, SO_EXCLUSIVEADDRUSE,
                         (char *) &iOptval, sizeof (iOptval));
    if (iResult == SOCKET_ERROR) {
        wprintf(L"setsockopt for SO_EXCLUSIVEADDRUSE failed with error: %ld\n",
               WSAGetLastError());
    }

    saLocal.sin_family = (ADDRESS_FAMILY) iFamily;
    saLocal.sin_port = htons( (u_short) iPort);
    saLocal.sin_addr.s_addr = htonl(INADDR_ANY);

    // Bind the socket
    iResult = bind(s, (SOCKADDR *) &saLocal, sizeof (saLocal));
    if (iResult == SOCKET_ERROR) {

        // Most errors related to setting SO_EXCLUSIVEADDRUSE
        // will occur at bind.
        iError = WSAGetLastError();
        if (iError == WSAEACCES)
            wprintf(L"bind failed with WSAEACCES (access denied)\n");
        else
            wprintf(L"bind failed with error: %ld\n", iError);

    } else {
        wprintf(L"bind on socket with SO_EXCLUSIVEADDRUSE succeeded to port: %ld\n",
               iPort);
    }

    // cleanup
    closesocket(s);
    WSACleanup();

    return 0;
}

```

To have any effect, the SO\_EXCLUSIVADDRUSE option must be set before the **bind** function is called (this also applies to the SO\_REUSEADDR option). Table 1 lists the effects of setting the SO\_EXCLUSIVEADDRUSE option. Wildcard indicates binding to the wildcard address, such as 0.0.0.0 for IPv4 and :: for IPv6. Specific indicates binding to a specific interface, such as binding an IP address assigned to an adapter. Specific2 indicates binding to a specific address other than the address bound to in the Specific case.

**NOTE**

The Specific2 case is applicable only when the first bind is performed with a specific address; for the case in which the first socket is bound to the wildcard, the entry for Specific covers all specific address cases.

For example, consider a computer with two IP interfaces: 10.0.0.1 and 10.99.99.99. If the first bind is to 10.0.0.1 and port 5150 with the SO\_EXCLUSIVEADDRUSE option set, then the second bind to 10.99.99.99 and port 5150 with any or no options set succeeds. However, if the first socket is bound to the wildcard address (0.0.0.0) and port 5150 with SO\_EXCLUSIVEADDRUSE set, any subsequent bind to the same port—regardless of IP address—will fail with either WSAEADDRINUSE (10048) or WSAEACCESS (10013), depending on which options were set on the second bind socket.

### Bind Behavior with Various Options Set

Second bind

First bind

*SO\_EXCLUSIVEADDRUSE*

*No options or SO\_REUSEADDR*

*Wildcard*

*Specific*

*Wildcard*

*Specific*

\${\{ROWSPAN3}\\$SO\_EXCLUSIVEADDRUSE\\${\REMOVE}\\$

*Wildcard*

Fail (10048)

Fail (10048)

Fail (10048)

Fail (10048)

*Specific*

Fail (10048)

Fail (10048)

Fail (10048)

Fail (10048)

*Specific2*

n/a

Success

n/a

Success

  \${{ROWSPAN3}}\$ *No options* \${{REMOVE}}\$

*Wildcard*

Fail (10048)

Fail (10048)

Fail (10048)

Fail (10048)

*Specific*

Fail (10048)

Fail (10048)

Fail (10048)

Fail (10048)

*Specific2*

n/a

Success

n/a

Success

  \${{ROWSPAN3}}\$ *SO\_REUSEADDR* \${{REMOVE}}\$

*Wildcard*

Fail (10013)

Fail (10013)

Success\*

Success

*Specific*

Fail (10013)

Fail (10013)

Success

Success\*

*Specific2*

n/a

Success

n/a

Success

\* The behavior is undefined as to which socket will receive packets.

In the case where the first bind sets no options or SO\_REUSEADDR, and the second bind performs a SO\_REUSEADDR, the second socket has overtaken the port and behavior regarding which socket will receive packets is undetermined. SO\_EXCLUSIVEADDRUSE was introduced to address this situation.

A socket with SO\_EXCLUSIVEADDRUSE set cannot always be reused immediately after socket closure. For example, if a listening socket with the exclusive flag set accepts a connection after which the listening socket is closed, another socket cannot bind to the same port as the first listening socket with the exclusive flag until the accepted connection is no longer active.

This situation can be quite complicated; even though the socket has been closed, the underlying transport may not terminate its connection. Even after the socket is closed, the system must send all of the buffered data, transmit a graceful disconnect to the peer, and wait for a graceful disconnect from the peer. It is therefore possible that the underlying transport may never release the connection, such as when the peer advertises a zero-size window, or other such attacks. In the previous example, the listening socket was closed after a client connection was accepted. Now even if the client connection is closed, the port still may not be reused if the client connection remains in an active state because of unacknowledged data, and so forth.

To avoid this situation, applications should ensure a graceful shutdown: call [shutdown](#) with the SD\_SEND flag, then wait in a [recv](#) loop until zero bytes are returned. Doing so avoids the problem associated with port reuse, guarantees all data was received by the peer, and assures the peer that all its data was successfully received.

The SO\_LINGER option may be set on a socket to prevent the port going into one of the active wait states; however, doing so is discouraged because it can lead to undesired effects, because it can cause the connection to be reset. For example, if data has been received but not yet acknowledged by the peer, and the local computer closes the socket with SO\_LINGER set, the connection is reset and the peer discards the unacknowledged data. Also, picking a suitable time to linger is difficult; a value too small results in many aborted connections, while a large timeout can leave the system vulnerable to denial of service attacks by establishing many connections, and thereby stalling numerous application threads.

**NOTE**

A socket that is using the SO\_EXCLUSIVEADDRUSE option must be shut down properly prior to closing it. Failure to do so can cause a denial of service attack if the associated service needs to restart.

## Requirements

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

REQUIREMENT	VALUE
Header	Winsock2.h

# SO\_KEEPALIVE socket option

3/5/2021 • 4 minutes to read • [Edit Online](#)

The **SO\_KEEPALIVE** socket option is designed to allow an application to enable keep-alive packets for a socket connection.

To query the status of this socket option, call the [getsockopt](#) function. To set this option, call the [setsockopt](#) function with the following parameters.

## Socket option value

The constant that represents this socket option is 0x0008.

## Syntax

```
int getsockopt(
    (SOCKET) s,      // descriptor identifying a socket
    (int) SOL_SOCKET, // level
    (int) SO_KEEPALIVE, // optname
    (char *) optval, // output buffer,
    (int) optlen, // size of output buffer
);
```

```
int setsockopt(
    (SOCKET) s,      // descriptor identifying a socket
    (int) SOL_SOCKET, // level
    (int) SO_KEEPALIVE, // optname
    (char *) optval, // input buffer,
    (int) optlen, // size of input buffer
);
```

## Parameters

*s* [in]

A descriptor identifying the socket.

*level* [in]

The level at which the option is defined. Use **SOL\_SOCKET** for this operation.

*optname* [in]

The socket option for which the value is to be set. Use **SO\_KEEPALIVE** for this operation.

*optval* [out]

A pointer to the buffer containing the value for the option to set. This parameter should point to buffer equal to or larger than the size of a **DWORD** value.

This value is treated as a boolean value with 0 used to indicate **FALSE** (disabled) and a nonzero value to indicate **TRUE** (enabled).

*optlen* [in, out]

A pointer to the size, in bytes, of the *optval*/buffer. This size must be equal to or larger than the size of a **DWORD** value.

## Return value

If the operation completes successfully, [setsockopt](#) returns zero.

If the operation fails, a value of **SOCKET\_ERROR** is returned and a specific error code can be retrieved by calling [WSAGetLastError](#).

ERROR CODE	MEANING
<a href="#">WSANOTINITIALISED</a>	A successful <a href="#">WSAStartup</a> call must occur before using this function.
<a href="#">WSAENETDOWN</a>	The network subsystem has failed.
<a href="#">WSAEFAULT</a>	One of the <i>optval</i> or the <i>optlen</i> parameters point to memory that is not in a valid part of the user address space. This error is also returned if the value pointed to by the <i>optlen</i> parameter is less than the size of a <b>DWORD</b> value.
<a href="#">WSAEINPROGRESS</a>	A blocking Windows Sockets 1.1 call is in progress, or the service provider is still processing a callback function.
<a href="#">WSAEINVAL</a>	The <i>level</i> parameter is unknown or invalid. On Windows Vista and later, this error is also returned if the socket was in a transitional state.
<a href="#">WSAENOPROTOOPT</a>	The option is unknown or unsupported by the indicated protocol family. This error is returned if the socket descriptor passed in the <i>s</i> parameter was for a datagram socket.
<a href="#">WSAENOTSOCK</a>	The descriptor is not a socket.

## Remarks

The [getsockopt](#) function called with the **SO\_KEEPALIVE** socket option allows an application to retrieve the current state of the keepalive option, although this is feature not normally used. If an application needs to enable keepalive packets on a socket, it justs calls the [setsockopt](#) function to enable the option.

The [setsockopt](#) function called with the **SO\_KEEPALIVE** socket option allows an application to enable keepalive packets for a socket connection. The **SO\_KEEPALIVE** option for a socket is disabled (set to **FALSE**) by default.

When this socket option is enabled, the TCP stack sends keep-alive packets when no data or acknowledgement packets have been received for the connection within an interval. For more information on the keep-alive option, see section 4.2.3.6 on the *Requirements for Internet Hosts—Communication Layers* specified in RFC 1122 available at the [IETF website](#). (This resource may only be available in English.)

The **SO\_KEEPALIVE** socket option is valid only for protocols that support the notion of keep-alive (connection-

oriented protocols). For TCP, the default keep-alive timeout is 2 hours and the keep-alive interval is 1 second. The default number of keep-alive probes varies based on the version of Windows.

The [SIO\\_KEEPALIVE\\_VALS](#) control code can be used to enable or disable keep-alive, and adjust the timeout and interval, for a single connection. If keep-alive is enabled with [SO\\_KEEPALIVE](#), then the default TCP settings are used for keep-alive timeout and interval unless these values have been changed using [SIO\\_KEEPALIVE\\_VALS](#).

The default system-wide value of the keep-alive timeout is controllable through the [KeepAliveTime](#) registry setting which takes a value in milliseconds. The default system-wide value of the keep-alive interval is controllable through the [KeepAliveInterval](#) registry setting which takes a value in milliseconds.

On Windows Vista and later, the number of keep-alive probes (data retransmissions) is set to 10 and cannot be changed.

On Windows Server 2003, Windows XP, and Windows 2000, the default setting for number of keep-alive probes is 5. The number of keep-alive probes is controllable through the [TcpMaxDataRetransmissions](#) and [PPTPTcpMaxDataRetransmissions](#) registry settings. The number of keep-alive probes is set to the larger of the two registry key values. If this number is 0, then keep-alive probes will not be sent. If this number is above 255, then it is adjusted to 255.

On Windows Vista and later, the [SO\\_KEEPALIVE](#) socket option can only be set using the [setsockopt](#) function when the socket is in a well-known state not a transitional state. For TCP, the [SO\\_KEEPALIVE](#) socket option should be set either before the connect function ([connect](#), [ConnectEx](#), [WSAConnect](#), [WSAConnectByList](#), or [WSAConnectByName](#)) is called, or after the connection request is actually completed. If the connect function was called asynchronously, then this requires waiting for the connection completion before trying to set the [SO\\_KEEPALIVE](#) socket option. If an application attempts to set the [SO\\_KEEPALIVE](#) socket option when a connection request is still in process, the [setsockopt](#) function will fail and return [WSAEINVAL](#).

On Windows Server 2003, Windows XP, and Windows 2000, the [SO\\_KEEPALIVE](#) socket option can be set using the [setsockopt](#) function when the socket is a transitional state (a connection request is still in progress) as well as a well-known state.

Note that the *Ws2def.h* header file is automatically included in *Winsock2.h*, and should never be used directly.

## Requirements

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Ws2def.h (include Winsock2.h)

## See also

[getsockopt](#)

[setsockopt](#)

[KeepAliveTime](#)

[KeepAliveInterval](#)

PPTPTcpMaxDataRetransmissions

socket

SIO\_KEEPALIVE\_VALS

TcpMaxDataRetransmissions

WSAGetLastError

# SO\_PORT\_SCALABILITY

3/5/2021 • 3 minutes to read • [Edit Online](#)

The SO\_PORT\_SCALABILITY socket option enables local port scalability for a socket.

## SO\_PORT\_SCALABILITY

0x3006

The SO\_PORT\_SCALABILITY socket option enables local port scalability by allowing port allocation to be maximized by allocating wildcard ports multiple times for different local address port pairs on a local machine.

## Remarks

Note: on platforms where both SO\_PORT\_SCALABILITY and SO\_REUSE\_UNICASTPORT are supported, prefer to use SO\_REUSE\_UNICASTPORT.

Proxy server environments have scalability issues because of limited local port availability. One way to work around this is to add more IP addresses to the machine. However, by default wildcard ports used with the **bind** function are limited to the size of the dynamic port range on the local machine (up to 64K ports, but usually less) no matter the number of IP addresses on the local machine. Working around this requires the application to maintain its own port pool either with port reservation or by using heuristics.

To avoid having every application that requires scalability manage its own port pool, and to allow for greater scalability while maintaining application compatibility, Windows Server 2008 introduced the SO\_PORT\_SCALABILITY socket option to help maximize wildcard port allocation. Port allocation is maximized by allowing an application to allocate wildcard ports for each unique local address and port pair. So if a local machine has four IP addresses, then up to 256 K wildcard ports (64 K ports × 4 IP addresses) can be allocated by wildcard **bind** function requests.

When the SO\_PORT\_SCALABILITY socket option is set on a socket and a call to the **bind** function is made for a specified address and wildcard port (the *name* parameter is set with a specific address and a port of 0), Winsock will allocate a port for the specified address. This allocation will be based on all of the possible IP addresses and ports/per address on the local computer. If a wildcard port is acquired using the SO\_PORT\_SCALABILITY option, that port cannot be allocated by another socket without the SO\_PORT\_SCALABILITY option. This restriction is in place to avoid backward-compatibility problems with applications that assume a wildcard local port cannot be reused. Note that this means that applications which acquire a large number of ports using SO\_PORT\_SCALABILITY can starve legacy applications of ports. If all available ephemeral ports have been acquired for at least one address with SO\_PORT\_SCALABILITY, then no more wildcard port allocations are possible without the socket option.

To have any effect, the SO\_PORT\_SCALABILITY option must be set before the **bind** function is called. An example of how this would be used on a computer with two addresses is outlined below:

- The **socket** function is called by a process to create a socket.
- The **setsockopt** function is called to enable the SO\_PORT\_SCALABILITY socket option on the newly created socket.
- The **bind** function is called to do a bind on one of the local computer's IP addresses and port 0.
- The **connect** function is then called to connect to a remote IP address. The socket is used by the application as needed.
- A **socket** function is called by the same process (possibly a different thread) to create a second socket.

- The [setsockopt](#) function is called to enable the **SO\_PORT\_SCALABILITY** socket option on the newly created second socket.
- The [bind](#) function is called with the local computer's second IP address and port 0. Even when all ports have been previously allocated, this call succeeds because there are multiple IP addresses available on the local computer and the **SO\_PORT\_SCALABILITY** socket option was set on both sockets in the same process.
- The [connect](#) function is then called to connect to a remote IP address. The second socket is used by the application as needed.

## Requirements

Requirement	Value
Minimum supported client	None supported
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	Ws2def.h

## See also

[getsockopt](#)

[setsockopt](#)

[SOL\\_SOCKET Socket Options](#)

[Socket Options](#)

# IP\_DSCP\_TRAFFIC\_TYPE

3/22/2021 • 2 minutes to read • [Edit Online](#)

The following table describes IP\_DSCP\_TRAFFIC\_TYPE Socket Options.

**\*\*IP\\_DSCP\\_TRAFFIC\\_TYPE\*\***

OPTION	GET	SET	OPTVAL TYPE	DESCRIPTION
DSCP_TRAFFIC_TYPE	Yes	Yes	DWORD	By setting this value to values defined in <b>DSCP_TRAFFIC_TYPE</b> , an application will be able to ask its network packets to be treated according to the type of service being requested. Similarly calls to <b>getsockopt</b> can be used to obtain the current setting for the type of traffic requested on the given socket

## Requirements

REQUIREMENT	VALUE
End of client support	Windows 10
End of server support	Windows Server 2016
Header	N/A

# Winsock IOCTLs

5/26/2021 • 37 minutes to read • [Edit Online](#)

This section describes Winsock Socket input/output controls (IOCTLs) for various editions of Windows operating systems. Use the [WSAIoctl](#) or [WSPIoctl](#) function to issue a Winsock IOCTL to control the mode of a socket, the transport protocol, or the communications subsystem.

Some Winsock IOCTLs require more explanation than this table can convey; such options contain links to additional topics.

It is possible to adopt an encoding scheme that preserves the currently defined [ioctlssocket](#) 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 built 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	O	V	T	VENDOR/ADDRESS FAMILY	CODE
3	3	2	2 2	2 2 2 2 2 2 1 1 1 1	1 1 1 1 1 1
1	0	9	8 7	6 5 4 3 2 1 0 9 8 7 6	5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

## NOTE

The bits in *dwIoControlCode* parameter displayed in the table must be read vertically from top to bottom by column. So the left-most bit is bit 31, the next bit is bit 30, and the right-most bit is bit 0.

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**. Control codes using both input and output buffers set both I and O.

V is set if there are no parameters for the code, as with **IOC\_VOID**.

T is a 2-bit quantity that defines the type of the 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, as with **IOC\_VENDOR**. This type allows companies to be assigned a vendor number that appears in the **Vendor/Address family** parameter. 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

parameter has the same value as the code on Unix. If this is a generic Windows Sockets 2 IOCTL ( $T == 1$ ) then this parameter can be used as an extension of the code parameter to provide additional code values.

**Code** The 16-bit quantity that contains the specific IOCTL code for the operation.

## Unix IOCTL codes

The following Unix IOCTL codes (commands) are supported.

### FIONBIO

Enable or disable non-blocking mode on socket  $s$ . The  $\text{lpvInBuffer}$  parameter points at an **unsigned long** (QoS), which is nonzero if non-blocking mode is to be enabled and zero if it is to be disabled. When a socket is created, it operates in blocking mode (that is, non-blocking mode is disabled). This is consistent with BSD sockets.

The [WSAAAsyncSelect](#) or [WSAEventSelect](#) routine automatically sets a socket to non-blocking mode. If [WSAAAsyncSelect](#) or [WSAEventSelect](#) has been issued on a socket, then any attempt to use [WSAOctl](#) 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 [WSAAAsyncSelect](#) by calling [WSAAAsyncSelect](#) with the  $\text{IEvent}$  parameter equal to zero, or disable [WSAEventSelect](#) by calling [WSAEventSelect](#) with the  $\text{INetworkEvents}$  parameter equal to zero.

### FIONREAD

Determine the amount of data that can be read atomically from socket  $s$ . The  $\text{lpvOutBuffer}$  parameter points at an **unsigned long** in which [WSAOctl](#) stores the result.

If the socket passed in the  $s$  parameter 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 a data stream is byte-oriented, this is not guaranteed).

If the socket passed in the  $s$  parameter is message oriented (for example, type SOCK\_DGRAM), **FIONREAD** returns the reports the total number of bytes available to read, not the size of the first datagram (message) queued on the socket.

### SIOCATMARK

Determine whether or not all OOB data has been read. This applies only to a socket of stream-style (for example, type SOCK\_STREAM) that has been configured for inline reception of any OOB data (SO\_OOBINLINE). If no OOB 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 [recv](#) operations will never mix OOB and normal data in the same call.)  $\text{lpvOutBuffer}$  points at a **BOOL** in which [WSAOctl](#) stores the result.

## Windows Sockets 2 commands

The following Windows Sockets 2 commands are supported.

### [SIO\\_ACQUIRE\\_PORT\\_RESERVATION](#) (opcode setting: I, T==3)

Request a runtime reservation for a block of TCP or UDP ports. For runtime port reservations, the port pool requires that reservations be consumed from the process on whose socket the reservation was granted.

Runtime port reservations last only as long as the lifetime of the socket on which the

[SIO\\_ACQUIRE\\_PORT\\_RESERVATION](#) IOCTL was called. In contrast, persistent port reservations created using the [CreatePersistentTcpPortReservation](#) or [CreatePersistentUdpPortReservation](#) function may be consumed by any process with the ability to obtain persistent reservations.

For more detailed information, see the [SIO\\_ACQUIRE\\_PORT\\_RESERVATION](#) reference.

**SIO\_ACQUIRE\_PORT\_RESERVATION** is supported on Windows Vista and later versions of the operating system.

#### **SIO\_ADDRESS\_LIST\_CHANGE (opcode setting: V, T==1)**

To receive notification of changes in the list of local transport addresses of the 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 through **SIO\_ADDRESS\_LIST\_QUERY**.

It is assumed (although not required) that the application uses overlapped I/O to be notified of change by completion of **SIO\_ADDRESS\_LIST\_CHANGE** request. Alternatively, if the **SIO\_ADDRESS\_LIST\_CHANGE** IOCTL is issued on a 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 through a call to **WSAEVENTSELECT** or **WSAAASYNCSOCKET** with FD\_ADDRESS\_LIST\_CHANGE bit set in the network event bitmask.

#### **SIO\_ADDRESS\_LIST\_QUERY (opcode setting: O, T==1)**

Obtains a list of local transport addresses of the socket's protocol family to which the application can bind. The list of addresses varies based on address family and some addresses are excluded from the list.

##### **NOTE**

In Windows Plug-n-Play environments, addresses can be added and 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 through the **SIO\_ADDRESS\_LIST\_CHANGE** IOCTL which provides for notification through either overlapped I/O 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 through overlapped I/O or by signaling FD\_ADDRESS\_LIST\_CHANGE event), the whole sequence of actions should be repeated.

For more detailed information, see the **SIO\_ADDRESS\_LIST\_QUERY** reference. **SIO\_ADDRESS\_LIST\_QUERY** is supported on Windows 2000 and later.

#### **SIO\_APPLY\_TRANSPORT\_SETTING (opcode setting: I, T==3)**

Applies a transport setting to a socket. The transport setting being applied is based on the **TRANSPORT\_SETTING\_ID** passed in the *lpvInBuffer* parameter.

The only transport setting currently defines is for the **REAL\_TIME\_NOTIFICATION\_CAPABILITY** capability on a TCP socket.

If the **TRANSPORT\_SETTING\_ID** passed has the **Guid** member set to **REAL\_TIME\_NOTIFICATION\_CAPABILITY**, then this is a request to apply real time notification settings for the TCP socket used with the **ControlChannelTrigger** to receive background network notifications in a Windows Store app.

For more detailed information, see the **SIO\_APPLY\_TRANSPORT\_SETTING** reference.

**SIO\_APPLY\_TRANSPORT\_SETTING** is supported on Windows 8, Windows Server 2012, and later.

#### **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 [Winsock Annexes](#) 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\_ASSOCIATE\_PORT\_RESERVATION (opcode setting: I, T==3)**

Associate a socket with a persistent or runtime reservation for a block of TCP or UDP ports identified by the port reservation token. The [SIO\\_ASSOCIATE\\_PORT\\_RESERVATION](#) IOCTL must be issued before the socket is bound. If and when the socket is bound, the port assigned to it will be selected from the port reservation identified by the given token. If no ports are available from the specified reservation, the [bind](#) function call will fail.

For more detailed information, see the [SIO\\_ASSOCIATE\\_PORT\\_RESERVATION](#) reference.

[SIO\\_ASSOCIATE\\_PORT\\_RESERVATION](#) is supported on Windows Vista and later versions of the operating system.

#### **SIO\_BASE\_HANDLE (opcode setting: O, T==1)**

Retrieves the base service provider handle for a given socket. The returned value is a [SOCKET](#).

A layered service provider would never intercept this IOCTL since the return value must be the socket handle from the base service provider.

If the output buffer is not large enough for a socket handle (the *cbOutBuffer* is less than the size of a [SOCKET](#)) or the *lpvOutBuffer* parameter is a **NULL** pointer, [SOCKET\\_ERROR](#) is returned as the result of this IOCTL and [WSAGetLastError](#) returns [WSAEFAULT](#).

[SIO\\_BASE\\_HANDLE](#) is defined in the *Mswsock.h* header file and supported on Windows Vista and later.

#### **SIO\_BSP\_HANDLE (opcode setting: O, T==1)**

Retrieves the base service provider handle for a socket used by the [WSASendMsg](#) function. The returned value is a [SOCKET](#).

This ioctl is used by a layered service provider to ensure the provider intercept the [WSASendMsg](#) function.

If the output buffer is not large enough for a socket handle (the *cbOutBuffer* is less than the size of a [SOCKET](#)) or the *lpvOutBuffer* parameter is a **NULL** pointer, [SOCKET\\_ERROR](#) is returned as the result of this IOCTL and [WSAGetLastError](#) returns [WSAEFAULT](#).

[SIO\\_BSP\\_HANDLE](#) is defined in the *Mswsock.h* header file and supported on Windows Vista and later.

#### **SIO\_BSP\_HANDLE\_SELECT (opcode setting: O, T==1)**

Retrieves the base service provider handle for a socket used by the [select](#) function. The returned value is a [SOCKET](#).

This ioctl is used by a layered service provider to ensure the provider intercept the [select](#) function.

If the output buffer is not large enough for a socket handle (the *cbOutBuffer* is less than the size of a [SOCKET](#)) or the *lpvOutBuffer* parameter is a **NULL** pointer, [SOCKET\\_ERROR](#) is returned as the result of this IOCTL and [WSAGetLastError](#) returns [WSAEFAULT](#).

**SIO\_BSP\_HANDLE\_SELECT** is defined in the *Mswsock.h* header file and supported on Windows Vista and later.

#### **SIO\_BSP\_HANDLE\_POLL (opcode setting: O, T==1)**

Retrieves the base service provider handle for a socket used by the **WSAPoll** function. The *lpOverlapped* parameter must be a **NULL** pointer. The returned value is a **SOCKET**.

This IOCTL is used by a layered service provider to ensure the provider intercept the **WSAPoll** function.

If the output buffer is not large enough for a socket handle (the *cbOutBuffer* is less than the size of a **SOCKET**), the *lpvOutBuffer* parameter is a **NULL** pointer, or the *lpOverlapped* parameter is not a **NULL** pointer, **SOCKET\_ERROR** is returned as the result of this IOCTL and **WSAGetLastError** returns **WSAEFAULT**.

**SIO\_BSP\_HANDLE\_POLL** is defined in the *Mswsock.h* header file and supported on Windows Vista and later.

#### **SIO\_CHK\_QOS (opcode setting: I, O, T==3)**

Retrieves information about QoS traffic characteristics. During the transitional phase on the sending system between flow setup and the receipt of a RESV message (see [How the RSVP Service Invokes TC](#) for more information on the transitional phase), traffic associated with an RSVP flow is shaped based on service type (**BEST EFFORT**, **CONTROLLED LOAD**, or **GUARANTEED**). For more information, see [Using SIO\\_CHK\\_QOS](#) in the [Quality of Service](#) section of the Platform SDK.

#### **SIO\_CPU\_AFFINITY (opcode setting: I, T==3)**

Enables port sharing and receive indication parallelization. When your application uses this socket option to associate sockets to different processors, and then binds the sockets to the same address, receive indications will be distributed across the sockets based on Receive Side Scaling (RSS) hash. The RSS settings don't change, so any given flow (local endpoint, remote endpoint pair) will always be indicated on the same processor. As a result, all packets belonging to a given flow will be indicated to the same socket. This IOCTL must be called prior to bind, otherwise **WSAEINVAL** will be returned. The input buffer is a processor index (0-based) of type **USHORT**. The IOCTL is incompatible with **SO\_REUSEADDR** and **SO\_REUSE\_MULTICASTPORT**. Only supported for UDP sockets.

##### **NOTE**

If you're targeting version 10.0.19041.0 (Windows 10, version 2004) of the Windows SDK, then use the value **0x98000015** instead of the name **SIO\_CPU\_AFFINITY**.

#### **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](#). This IOCTL is not supported for IPv6 sockets and returns the [WSAENOPROTOOPT](#) error code.

### **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.

The GUID values for extension functions supported by the Windows TCP/IP service provider are defined in the *Mswsock.h* header file. The possible value for these GUIDs are as follows:

TERM	DESCRIPTION
WSAID_ACCEPTEX	The <a href="#">AcceptEx</a> extension function.
WSAID_CONNECTEX	The <a href="#">ConnectEx</a> extension function.
WSAID_DISCONNECTEX	The <a href="#">DisconnectEx</a> extension function.
WSAID_GETACCEPTEXSOCKADDRS	The <a href="#">GetAcceptExSockaddrs</a> extension function.
WSAID_TRANSMITFILE	The <a href="#">TransmitFile</a> extension function.
WSAID_TRANSMITPACKETS	The <a href="#">TransmitPackets</a> extension function.
WSAID_WSARECVMSG	The <a href="#">LPFN_WSARECVMSG (WSARecvMsg)</a> extension function.
WSAID_WSASENDMSG	The <a href="#">WSASendMsg</a> extension function.

### **SIO\_GET\_GROUP\_QOS (opcode setting: O, I, T==1)**

Reserved for future use with sockets.

Retrieve the [QOS](#) structure associated with the socket group to which this socket belongs. The input buffer is optional. Some protocols (for example, RSVP) allow the input buffer to be used to qualify a quality of service request. The [QOS](#) structure will be copied into the output buffer. If this socket does not belong to an appropriate socket group, the [SendingFlowspec](#) and [ReceivingFlowspec](#) members of the returned [QOS](#) structure are set to [NULL](#). The [WSAENOPROTOOPT](#) error code is indicated for service providers that do not support quality of service.

### **SIO\_GET\_INTERFACE\_LIST (opcode setting: O, T==0)**

Returns a list of configured IP interfaces and their parameters as an array of [INTERFACE\\_INFO](#) structures.

#### **NOTE**

Support of this command is mandatory for Windows Sockets 2-compliant TCP/IP service providers.

The *lpvOutBuffer* parameter points to the buffer in which to store the information about interfaces as an array of [INTERFACE\\_INFO](#) structures for unicast IP addresses on the interfaces. The *cbOutBuffer* parameter specifies the length of the output buffer. The number of interfaces returned (number of structures returned in the buffer pointed to by *lpvOutBuffer* parameter) can be determined based on the actual length of the output buffer

returned in *lpcbBytesReturned* parameter.

If the **WSAIoctl** function is called with SIO\_GET\_INTERFACE\_LIST and the level member of the socket *s* parameter is not defined as IPPROTO\_IP, WSAEINVAL is returned. A call to the **WSAIoctl** function with SIO\_GET\_INTERFACE\_LIST returns WSAEFAULT if the *cbOutBuffer* parameter that specifies the length of the output buffer is too small to receive the list of configured interfaces.

SIO\_GET\_INTERFACE\_LIST is supported on Windows Me/98 and Windows NT 4.0 with SP4 and later.

#### **SIO\_GET\_INTERFACE\_LIST\_EX (opcode setting: O, T==0)**

Reserved for future use with sockets.

Returns a list of configured IP interfaces and their parameters as an array of **INTERFACE\_INFO\_EX** structures.

The *lpvOutBuffer* parameter points to the buffer in which to store the information about interfaces as an array of **INTERFACE\_INFO\_EX** structures for unicast IP addresses on the interface. The *cbOutBuffer* parameter specifies the length of the output buffer. The number of interfaces returned (number of structures returned in *lpvOutBuffer*) can be determined based on the actual length of the output buffer returned in *lpcbBytesReturned* parameter.

SIO\_GET\_INTERFACE\_LIST\_EX is not currently supported on Windows.

#### **SIO\_GET\_QOS (opcode setting: O, T==1)**

Reserved for future use with sockets. Retrieve the **QOS** structure associated with the socket. The input buffer is optional. Some protocols (for example, RSVP) allow the input buffer to be used to qualify a quality of service request. The **QOS** structure will be copied into the output buffer. The output buffer must be sized large enough to be able to contain the full **QOS** structure. The **WSAENOPROTOOPT** error code is indicated for service providers that do not support quality of service.

A sender may not call **SIO\_GET\_QOS** until the socket is connected.

A receiver may call **SIO\_GET\_QOS** as soon as it is bound.

#### **SIO\_GET\_TX\_TIMESTAMP**

A socket IOCTL used to get timestamps for transmitted (TX) packets. Valid only for datagram sockets.

The **SIO\_GET\_TX\_TIMESTAMP** control code removes a transmit timestamp from a socket's transmit timestamp queue. Enable timestamp reception first by using the **SIO\_TIMESTAMPING** socket IOCTL. Then retrieve tx timestamps by ID by calling the **WSAIoctl** (or **WSPIoctl**) function with the following parameters.

For **SIO\_GET\_TX\_TIMESTAMP**, the input is a **UINT32** timestamp ID, and the output is a **UINT64** timestamp value. On success, the tx timestamp is available, and is returned. If no transmit timestamps are available, then **WSAGetLastError** returns **WSAEWOULDBLOCK**.

##### **NOTE**

TX timestamps are not supported when doing a coalesced send via **UDP\_SEND\_MSG\_SIZE**.

Also see [Winsock timestamping](#).

#### **SIO\_IDEAL\_SEND\_BACKLOG\_CHANGE (opcode setting: V, T==0)**

Notifies an application when the ideal send backlog (ISB) value changes for the underlying connection.

When sending data over a TCP connection using Windows sockets, it is important to keep a sufficient amount of data outstanding (sent but not acknowledged yet) in TCP in order to achieve the highest throughput. The ideal value for the amount of data outstanding to achieve the best throughput for the TCP connection is called the ideal send backlog (ISB) size. The ISB value is a function of the bandwidth-delay product of the TCP connection

and the receiver's advertised receive window (and partly the amount of congestion in the network).

The ISB value per connection is available from the TCP protocol implementation in Windows Server 2008, Windows Vista with SP1, and later versions of the operating system. The **SIO\_IDEAL\_SEND\_BACKLOG\_CHANGE** IOCTL can be used by an application to get notification when the ISB value changes dynamically for a connection.

For more detailed information, see the [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#) reference.

**SIO\_IDEAL\_SEND\_BACKLOG\_CHANGE** is supported on Windows Server 2008, Windows Vista with SP1, and later versions of the operating system.

#### **SIO\_IDEAL\_SEND\_BACKLOG\_QUERY (opcode setting: O, T==0)**

Retrieves the ideal send backlog (ISB) value for the underlying connection.

When sending data over a TCP connection using Windows sockets, it is important to keep a sufficient amount of data outstanding (sent but not acknowledged yet) in TCP in order to achieve the highest throughput. The ideal value for the amount of data outstanding to achieve the best throughput for the TCP connection is called the ideal send backlog (ISB) size. The ISB value is a function of the bandwidth-delay product of the TCP connection and the receiver's advertised receive window (and partly the amount of congestion in the network).

The ISB value per connection is available from the TCP protocol implementation in Windows Server 2008 and later. The **SIO\_IDEAL\_SEND\_BACKLOG\_QUERY** IOCTL can be used by an application to query the ISB value for a connection.

For more detailed information, see the [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#) reference.

**SIO\_IDEAL\_SEND\_BACKLOG\_QUERY** is supported on Windows Server 2008, Windows Vista with SP1, and later versions of the operating system.

#### **SIO\_KEEPALIVE\_VALS (opcode setting: I, T==3)**

Enables or disables the per-connection setting of the TCP **keep-alive** option which specifies the TCP keep-alive timeout and interval. For more information on the keep-alive option, see section 4.2.3.6 on the *Requirements for Internet Hosts—Communication Layers* specified in RFC 1122 available at the [IETF website](#). (This resource may only be available in English.)

**SIO\_KEEPALIVE\_VALS** can be used to enable or disable keep-alive probes and set the keep-alive timeout and interval. The keep-alive timeout specifies the timeout, in milliseconds, with no activity until the first keep-alive packet is sent. The keep-alive interval specifies the interval, in milliseconds, between when successive keep-alive packets are sent if no acknowledgement is received.

The **SO\_KEEPALIVE** option, which is one of the [SOL\\_SOCKET Socket Options](#), can also be used to enable or disable the TCP keep-alive on a connection, as well as query the current state of this option. To query whether TCP keep-alive is enabled on a socket, the **getsockopt** function can be called with the **SO\_KEEPALIVE** option. To enable or disable TCP keep-alive, the **setsockopt** function can be called with the **SO\_KEEPALIVE** option. If TCP keep-alive is enabled with **SO\_KEEPALIVE**, then the default TCP settings are used for keep-alive timeout and interval unless these values have been changed using **SIO\_KEEPALIVE\_VALS**.

For more detailed information, see the [SIO\\_KEEPALIVE\\_VALS](#) reference. **SIO\_KEEPALIVE\_VALS** is supported on Windows 2000 and later.

#### **SIO\_LOOPBACK\_FAST\_PATH (opcode setting: I, T==3)**

Configures a TCP socket for lower latency and faster operations on the loopback interface. This IOCTL requests that the TCP/IP stack uses a special fast path for loopback operations on this socket. The **SIO\_LOOPBACK\_FAST\_PATH** IOCTL can be used only with TCP sockets. This IOCTL must be used on both sides of the loopback session. The TCP loopback fast path is supported using either the IPv4 or IPv6 loopback interface. By default, **SIO\_LOOPBACK\_FAST\_PATH** is disabled.

For more detailed information, see the [SIO\\_LOOPBACK\\_FAST\\_PATH](#) reference. SIO\_LOOPBACK\_FAST\_PATH is supported on Windows 8, Windows Server 2012, and later.

#### **SIO\_MULTIPOINT\_LOOPBACK (opcode setting: V, T==1)**

Controls whether data sent by an application on the local computer (not necessarily by the same socket) in a multicast session will be received by a socket joined to the multicast destination group on the loopback interface. A value of **TRUE** causes multicast data sent by an application on the local computer to be delivered to a listening socket on the loopback interface. A value of **FALSE** prevents multicast data sent by an application on the local computer from being delivered to a listening socket on the loopback interface. By default, SIO\_MULTIPOINT\_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\_QUERY\_RSS\_PROCESSOR\_INFO (opcode setting: O, T==1)**

Queries the association between a socket and an RSS processor core and NUMA node.

The [SIO\\_QUERY\\_RSS\\_PROCESSOR\\_INFO](#) IOCTL returns a [SOCKET\\_PROCESSOR\\_AFFINITY](#) structure that contains the [PROCESSOR\\_NUMBER](#) and the NUMA node ID. The returned [PROCESSOR\\_NUMBER](#) structure contains a group number and relative processor number within the group.

For more detailed information, see the [SIO\\_QUERY\\_RSS\\_PROCESSOR\\_INFO](#) reference.

SIO\_QUERY\_RSS\_PROCESSOR\_INFO is supported on Windows 8, Windows Server 2012, and later.

#### **SIO\_QUERY\_RSS\_SCALABILITY\_INFO (opcode setting: O, T==3)**

Queries offload interfaces for receive-side scaling (RSS) capability. The argument structure returned for SIO\_QUERY\_RSS\_SCALABILITY\_INFO is specified in the [RSS\\_SCALABILITY\\_INFO](#) structure defined in the *Mstcpip.h* header file. This structure is defined as follows:

```
// Scalability info for the transport
typedef struct _RSS_SCALABILITY_INFO {
    BOOLEAN RssEnabled;
} RSS_SCALABILITY_INFO, *PRSS_SCALABILITY_INFO;
```

The value returned in the **RssEnabled** member indicates if RSS is enabled on at least one interface.

If the output buffer is not large enough for the [RSS\\_SCALABILITY\\_INFO](#) structure (the *cbOutBuffer* is less than the size of a [RSS\\_SCALABILITY\\_INFO](#)) or the *lpvOutBuffer* parameter is a **NULL** pointer, [SOCKET\\_ERROR](#) is returned as the result of this IOCTL and [WSAGetLastError](#) returns [WSAEINVAL](#).

In high-speed networking where multiple CPUs reside within a single system, the ability of the networking protocol stack to scale well on a multi-CPU system is inhibited because the architecture of NDIS 5.1 and earlier versions limits receive protocol processing to a single CPU. Receive-side scaling (RSS) resolves this issue by allowing the network load from a network adapter to be balanced across multiple CPUs.

SIO\_QUERY\_RSS\_SCALABILITY\_INFO is supported on Windows Vista and later.

#### **SIO\_QUERY\_TRANSPORT\_SETTING (opcode setting: I, T==3)**

Queries the transport settings on a socket. The transport setting being queried is based on the [TRANSPORT\\_SETTING\\_ID](#) passed in the *lpvInBuffer* parameter.

The only transport setting currently defines is for the **REAL\_TIME\_NOTIFICATION\_CAPABILITY** capability on a TCP socket.

If the **TRANSPORT\_SETTING\_ID** has the **Guid** member set to **REAL\_TIME\_NOTIFICATION\_CAPABILITY**, then this is a request to query the real time notification settings for the TCP socket used with the **ControlChannelTrigger** to receive background network notifications in a Windows Store app. If the **WSAIoctl** or **WSPIoctl** call is successful, this IOCTL returns a **REAL\_TIME\_NOTIFICATION\_SETTING\_OUTPUT** structure with the current status.

For more detailed information, see the [SIO\\_QUERY\\_TRANSPORT\\_SETTING](#) reference.

**SIO\_QUERY\_TRANSPORT\_SETTING** is supported on Windows 8, Windows Server 2012, and later.

#### **SIO\_QUERY\_WFP\_ALE\_ENDPOINT\_HANDLE (opcode setting: O, T==3)**

Queries the Application Layer Enforcement (ALE) endpoint handle.

The Windows Filtering Platform (WFP) supports network traffic inspection and modification. On Windows Vista, WFP focuses on scenarios where the host machine is the communication endpoint. On Windows Server 2008, however, there are edge firewall implementations which would like to leverage the WFP platform to inspect and proxy pass-through traffic. The Internet Security and Acceleration (ISA) server is an example of such an edge device.

There are some firewall scenarios that may require the ability to inject an inbound packet into the send path associated with an existing endpoint. There needs to be a mechanism to discover the transport layer endpoint handle associated with the destination endpoint. The application that created the endpoint owns these transport layer endpoints. This IOCTL is used to provide socket handle to transport layer endpoint handle mapping.

If the output buffer is not large enough for the endpoint handle (the *cbOutBuffer* is less than the size of a **UINT64**) or the *lpvOutBuffer* parameter is a **NULL** pointer, **SOCKET\_ERROR** is returned as the result of this IOCTL and **WSAGetLastError** returns **WSAEINVAL**.

**SIO\_QUERY\_WFP\_ALE\_ENDPOINT\_HANDLE** is supported on Windows Vista and later.

#### **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_CONTEXT (opcode setting: I, T==3)**

Queries the redirect context for a redirect record used by a Windows Filtering Platform (WFP) redirect service.

The **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_CONTEXT** IOCTL is used to provide proxied connection tracking on redirected socket connections. This WFP feature facilitates tracking of redirection records from the initial redirect of a connection to the final connection to the destination.

For more detailed information, see the [SIO\\_QUERY\\_WFP\\_CONNECTION\\_REDIRECT\\_CONTEXT](#) reference.

**SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_CONTEXT** is supported on Windows 8, Windows Server 2012, and later.

#### **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_RECORDS (opcode setting: I, T==3)**

Queries the redirect record for the accepted TCP/IP connection for use by a Windows Filtering Platform (WFP) redirect service.

The **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_RECORDS** IOCTL is used to provide proxied connection tracking on redirected socket connections. This WFP feature facilitates tracking of redirection records from the initial redirect of a connection to the final connection to the destination.

For more detailed information, see the [SIO\\_QUERY\\_WFP\\_CONNECTION\\_REDIRECT\\_RECORDS](#) reference.

**SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_RECORDS** is supported on Windows 8, Windows Server 2012, and later.

#### **SIO\_RECVALL (opcode setting: I, T==3)**

Enables a socket to receive all IPv4 or IPv6 packets passing through a network interface. The socket handle

passed to the [WSAO ioctl](#) function must be one of the following:

- An IPv4 socket that was created with the address family set to AF\_INET, the socket type set to SOCK\_RAW, and the protocol set to IPPROTO\_IP.
- An IPv6 socket that was created with the address family set to AF\_INET6, the socket type set to SOCK\_RAW, and the protocol set to IPPROTO\_IPV6.

The socket also must be bound to an explicit local IPv4 or IPv6 interface, which means that you cannot bind to INADDR\_ANY or in6addr\_any.

On Windows Server 2008 and earlier, the [SIO\\_RCVALL](#) IOCTL setting would not capture local packets sent out of a network interface. This included packets received on another interface and forwarded out the network interface specified for the SIO\_RCVALL IOCTL.

On Windows 7 and Windows Server 2008 R2 , this was changed so that local packets sent out of a network interface are also captured. This includes packets received on another interface and then forwarded out the network interface bound to the socket with [SIO\\_RCVALL](#) IOCTL.

Setting this IOCTL requires Administrator privilege on the local computer.

This feature is sometimes referred to as promiscuous mode.

The possible values for the SIO\_RCVALL IOCTL option are specified in the RCVALL\_VALUE enumeration defined in the *Mstcpip.h* header file. The possible values for SIO\_RCVALL are as follows:

TERM	DESCRIPTION
RCVALL_OFF	Disable this option so a socket does not receive all IPv4 or IPv6 packets on the network.
RCVALL_ON	Enable this option so a socket receives all IPv4 or IPv6 packets on the network. This option enables promiscuous mode on the network interface card (NIC), if the NIC supports promiscuous mode. On a LAN segment with a network hub, a NIC that supports promiscuous mode will capture all IPv4 or IPv6 traffic on the LAN, including traffic between other computers on the same LAN segment. All of the captured packets (IPv4 or IPv6, depending on the socket) will be delivered to the raw socket. This option will not capture other packets (ARP, IPX, and NetBEUI packets, for example) on the interface. Netmon uses the same mode for the network interface, but does not use this option to capture traffic.
RCVALL_SOCKETLEVELONLY	This feature is not currently implemented, so setting this option does not have any affect.
RCVALL_IPLEVEL	Enable this option so an IPv4 or IPv6 socket receives all packets at the IP level on the network. This option does not enable promiscuous mode on the network interface card. This option only affects packet processing at the IP level. The NIC still receives only packets directed to its configured unicast and multicast addresses. However, a socket with this option enabled will receive not only packets directed to specific IP addresses, but will receive all the IPv4 or IPv6 packets the NIC receives. This option will not capture other packets (ARP, IPX, and NetBEUI packets, for example) received on the interface.

For more detailed information, see the [SIO\\_RECVALL](#) reference.

SIO\_RECVALL is supported on Windows 2000 and later.

#### **SIO\_RECVALL\_IGMPMCCAST (opcode setting: I, T==3)**

Enables a socket to receive all IGMP multicast IP traffic on the network, without receiving other multicast IP traffic. The socket handle passed to the [WSAOctl](#) function must be of AF\_INET address family, SOCK\_RAW socket type, and IPPROTO\_IGMP protocol. The socket also must be bound to an explicit local interface, which means that you cannot bind to INADDR\_ANY.

Once the socket is bound and the IOCTL set, calls to the [WSARecv](#) or [recv](#) functions return multicast IP datagrams passing through the given interface. Note that you must supply a sufficiently large buffer. Setting this IOCTL requires Administrator privilege on the local computer.

SIO\_RECVALL\_IGMPMCCAST is supported on Windows 2000 and later.

#### **SIO\_RECVALL\_MCAST (opcode setting: I, T==3)**

Enables a socket to receive all multicast IP traffic on the network (that is, all IP packets destined for IP addresses in the range of 224.0.0.0 to 239.255.255.255). The socket handle passed to the [WSAOctl](#) function must be of AF\_INET address family, SOCK\_RAW socket type, and IPPROTO\_UDP protocol. The socket also must bind to an explicit local interface, which means that you cannot bind to INADDR\_ANY. The socket should bind to port zero.

Once the socket is bound and the IOCTL set, calls to the [WSARecv](#) or [recv](#) functions return multicast IP datagrams passing through the given interface. Note that you must supply a sufficiently large buffer. Setting this IOCTL requires Administrator privilege on the local computer.

SIO\_RECVALL\_MCAST is supported on Windows 2000 and later.

#### **SIO\_RELEASE\_PORT\_RESERVATION (opcode setting: I, T==3)**

Releases a runtime reservation for a block of TCP or UDP ports. The runtime reservation to be released must have been obtained from the issuing process using the [SIO\\_ACQUIRE\\_PORT\\_RESERVATION](#) IOCTL.

For more detailed information, see the [SIO\\_RELEASE\\_PORT\\_RESERVATION](#) reference.

SIO\_RELEASE\_PORT\_RESERVATION is supported on Windows Vista and later versions of the operating system.

#### **SIO\_ROUTING\_INTERFACE\_CHANGE (opcode setting: I, T==1)**

To receive notification of a routing interface change that should be used to reach the remote address in the input buffer (specified as a [sockaddr](#) structure). No output information on the new routing interface will be provided upon completion of this IOCTL; the completion merely indicates that the routing interface for a given destination has changed and should be queried using the [SIO\\_ROUTING\\_INTERFACE\\_QUERY](#) IOCTL.

It is assumed, although not required, that the application uses overlapped I/O to be notified of the routing interface change through completion of SIO\_ROUTING\_INTERFACE\_CHANGE request. Alternatively, if the SIO\_ROUTING\_INTERFACE\_CHANGE IOCTL is issued on a non-blocking socket with the *lpOverlapped* and *lpCompletionRoutine* parameters set to **NULL**, it will complete immediately returning and [WSAEWOULDBLOCK](#) as an error, and the application can then wait for routing change events through call to [WSAEVENTSELECT](#) or [WSAAASYNCSOCKET](#) 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 the service provider keep track of these notification requests will use a significant amount 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 and destinations it has and then use SIO\_ROUTING\_INTERFACE\_QUERY to get the actual routing information.

- A service provider has the option to ignore the information specified 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\_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 through the SIO\_ROUTING\_INTERFACE\_CHANGE IOCTL which provides for notification through either overlapped I/O or a 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 through overlapped I/O or by signaling FD\_ROUTING\_INTERFACE\_CHANGE event), the whole sequence of actions should be repeated.

If the 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 `/pcbBytesReturned` in this case. Note the WSAEFAULT error code is also returned if the `/pvInBuffer`, `/pvOutBuffer`, or `/pcbBytesReturned` 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 through 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\_SET\_COMPATIBILITY\_MODE (opcode setting: I, T==3)

Requests how the networking stack should handle certain behaviors for which the default way of handling the behavior may differ across Windows versions. The argument structure for SIO\_SET\_COMPATIBILITY\_MODE is specified in the `WSA_COMPATIBILITY_MODE` structure defined in the `Mswsockdef.h` header file. This structure is defined as follows:

```
/* Argument structure for SIO_SET_COMPATIBILITY_MODE */
typedef struct _WSA_COMPATIBILITY_MODE {
    WSA_COMPATIBILITY_BEHAVIOR_ID BehaviorId;
    ULONG TargetOsVersion;
} WSA_COMPATIBILITY_MODE, *PWSA_COMPATIBILITY_MODE;
```

The value specified in the `BehaviorId` member indicates the behavior requested. The value specified in the `TargetOsVersion` member indicates the Windows version that is being requested for the behavior.

The `BehaviorId` member can be one of the values from the `WSA_COMPATIBILITY_BEHAVIOR_ID` enumeration type defined in the `Mswsockdef.h` header file. The possible values for the `BehaviorId` member are as follows.

TERM	DESCRIPTION
WsaBehaviorAll	This is equivalent to requesting all of the possible compatible behaviors defined for <b>WSA_COMPATIBILITY_BEHAVIOR_ID</b> .
WsaBehaviorReceiveBuffering	When the <b>TargetOsVersion</b> member is set to a value for Windows Vista or later, reductions to the TCP receive buffer size on this socket using the <b>SO_RCVBUF</b> socket option are allowed even after a TCP connection has been established. When the <b>TargetOsVersion</b> member is set to a value earlier than Windows Vista, reductions to the TCP receive buffer size on this socket using the <b>SO_RCVBUF</b> socket option are not allowed after connection establishment.
WsaBehaviorAutoTuning	When the <b>TargetOsVersion</b> member is set to a value for Windows Vista or later, receive window auto-tuning is enabled and the TCP window scale factor is reduced to 2 from the default value of 8. When the <b>TargetOsVersion</b> is set to a value earlier than Windows Vista, receive window auto-tuning is disabled. The TCP window scaling option is also disabled and the maximum true receive window size is limited to 65,535 bytes. The TCP window scaling option can't be negotiated on the connection even if the <b>SO_RCVBUF</b> socket option was called on this socket specifying a value greater than 65,535 bytes before the connection was established.

For more detailed information, see the [SIO\\_SET\\_COMPATIBILITY\\_MODE](#) reference.

**SIO\_SET\_COMPATIBILITY\_MODE** is supported on Windows Vista and later.

#### **SIO\_SET\_GROUP\_QOS (opcode setting: I, T==1)**

Reserved.

#### **SIO\_SET\_PRIORITY\_HINT (opcode setting: I, T==3)**

Provides a hint to the underlying transport protocol to treat the traffic on this socket with a specific priority. The *IpvInBuffer* must point to a variable of type **PRIORITY\_HINT** with *cblnBuffer* set to *sizeof(PRIORITY\_HINT)*. The *IpvOutBuffer* and *cbOutBuffer* parameters must be **NULL** and 0, respectively. The Microsoft Windows TCP implementation supports this IOCTL starting with Windows 10, version 1809 (10.0; Build 17763) and later as follows: when the requested priority value is set to **IoPriorityHintVeryLow**, TCP uses a modified version of the LEDBAT algorithm (defined in RFC 6817) for controlling the outbound traffic rate on the socket. The inbound traffic is not affected by this IOCTL. LEDBAT is a scavenger algorithm, and its goal is to keep latency low and prevent any adverse effect on normal-priority traffic by getting out of the way when normal-priority traffic is present.

Also see [RFC 6817](#).

**SIO\_SET\_PRIORITY\_HINT** is supported on Windows 10, version 1809 (10.0; Build 17763) and later.

#### **SIO\_SET\_QOS (opcode setting: I, T==1)**

Associate the specified **QOS** structure with the socket. No output buffer is required, the **QOS** structure will be obtained from the input buffer. The [WSAENOPROTOOPT](#) error code is indicated for service providers that do not support quality of service.

#### **SIO\_TCP\_INITIAL\_RTO (opcode setting: I, T==3)**

Controls the initial (SYN / SYN+ACK) retransmission characteristics of a TCP socket by configuring initial

retransmission timeout (RTO) parameters. The configuration parameters are specified in a [TCP\\_INITIAL\\_RTO\\_PARAMETERS](#) structure.

For more detailed information, see the [SIO\\_TCP\\_INITIAL\\_RTO](#) reference. [SIO\\_TCP\\_INITIAL\\_RTO](#) is supported on Windows 8, Windows Server 2012, and later.

## SIO\_TIMESTAMPING

A socket IOCTL used to configure reception of socket transmit/receive timestamps. Valid only for datagram sockets. The input type for SIO\_TIMESTAMPING is the [TIMESTAMPING\\_CONFIG](#) structure.

Also see [Winsock timestamping](#).

## 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 [Winsock Annexes](#) 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 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 backward compatibility with systems where COM is not available or cannot be used for some other reason.

## SIO\_UDP\_CONNRESET (opcode setting: I, T==3)

**Windows XP:** Controls whether UDP PORT\_UNREACHABLE messages are reported. Set to TRUE to enable reporting. Set to FALSE to disable reporting.

## SIO\_SET\_WFP\_CONNECTION\_REDIRECT\_RECORDS (opcode setting: I, T==3)

Sets the redirect record to the new TCP socket used for connecting to the final destination for use by a Windows Filtering Platform (WFP) redirect service.

The [SIO\\_SET\\_WFP\\_CONNECTION\\_REDIRECT\\_RECORDS](#) IOCTL is used as part of proxied connection tracking on redirected socket connections. This WFP feature facilitates tracking of redirection records from the initial redirect of a connection to the final connection to the destination.

For more detailed information, see the [SIO\\_SET\\_WFP\\_CONNECTION\\_REDIRECT\\_RECORDS](#) reference. [SIO\\_SET\\_WFP\\_CONNECTION\\_REDIRECT\\_RECORDS](#) is supported on Windows 8, Windows Server 2012, and later.

## SIO\_TCP\_INFO (opcode setting: I, O, T==3)

Retrieves the TCP statistics for a socket. The TCP statistics are provided in a [TCP\\_INFO\\_V0](#) structure.

Unlike retrieving TCP statistics with the [GetPerTcpConnectionEStats](#) function, retrieving TCP statistics with this control code does not require the user code to load, store, and filter the TCP connection table, and does not require elevated privileges to use.

For more information, see [SIO\\_TCP\\_INFO](#). [SIO\\_TCP\\_INFO](#) is supported on Windows 10, version 1703, Windows Server 2016, and later.

## Remarks

Winsock IOCTLs are defined in a number of different header files. These include the *Winsock2.h*, *Mswsock.h*, and *Mstcpip.h* header file.

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the

organization of header files has changed and a number of Winsock loctls are also defined in the *Ws2def.h*, *Ws2ipdef.h*, and *Mswsockdef.h* header files. The *Ws2def.h* header file is automatically included by the *Winsock2.h* header file. The *Ws2ipdef.h* header file is automatically included by the *Ws2tcpip.h* header file. The *Mswsockdef.h* header file is automatically included in the *Mswsockdef.h* header file.

## Requirements

Requirement	Value
Header	Winsock2.h; Mstcpip.h; Mswsock.h; Mswsockdef.h on Windows Vista, Windows Server 2008 and Windows 7 (include Mswsock.h); Ws2def.h on Windows Vista, Windows Server 2008 and Windows 7 (include Winsock2.h); Ws2ipdef.h on Windows Vista, Windows Server 2008 and Windows 7 (include Ws2tcpip.h)

# SIO\_ACQUIRE\_PORT\_RESERVATION control code

3/5/2021 • 10 minutes to read • [Edit Online](#)

## Description

The SIO\_ACQUIRE\_PORT\_RESERVATION control code acquires a runtime reservation for a block of TCP or UDP ports.

To perform this operation, call the [WSAIoctl](#) or [WSPIoctl](#) function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_ACQUIRE_PORT_RESERVATION, // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to an INET_PORT_RANGE structure
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    (LPVOID) lpvOutBuffer, // pointer to an INET_PORT_RESERVATION_INSTANCE structure
    (DWORD) cbOutBuffer,  // size, in bytes, of the output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_ACQUIRE_PORT_RESERVATION, // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to an INET_PORT_RANGE structure
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    (LPVOID) lpvOutBuffer, // pointer to a INET_PORT_RESERVATION_INSTANCE structure
    (DWORD) cbOutBuffer,  // size, in bytes, of the output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);
```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use SIO\_ACQUIRE\_PORT\_RESERVATION for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter contains a pointer to an [INET\\_PORT\\_RANGE](#) structure that specifies the starting point number and the number of ports to reserve.

### cbInBuffer

The size, in bytes, of the input buffer. This parameter should be the size of the [INET\\_PORT\\_RANGE](#) structure.

### lpvOutBuffer

A pointer to the output buffer. On successful output, this parameter contains a pointer to an

[INET\\_PORT\\_RESERVATION\\_INSTANCE](#) structure.

#### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter must be at least the size of the [INET\\_PORT\\_RESERVATION\\_INSTANCE](#) structure.

#### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer.

If the output buffer is too small, the call fails, [WSAGetLastError](#) returns [WSAEINVAL](#), and the *lpcbBytesReturned* parameter points to a **DWORD** value of zero.

If *lpOverlapped* is **NULL**, the **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned on a successful call cannot be zero.

If the *lpOverlapped* parameter is not **NULL** for overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned may be zero since the size of the data stored can't be determined until the overlapped operation has completed. The final completion status can be retrieved when the appropriate completion method is signaled when the operation has completed.

#### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAOctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

#### **IpCompletionRoutine**

Type: \_In\_opt\_ [LPWSAOVERLAPPED\\_COMPLETION\\_ROUTINE](#)

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

#### **IpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

#### **IpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAOctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAOctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
<b>WSA_IO_PENDING</b>	Overlapped I/O operation is in progress. This value is returned if an overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSA_OPERATION_ABORTED</b>	The I/O operation has been aborted because of either a thread exit or an application request. This error is returned if an overlapped operation was canceled due to the closure of the socket or the execution of the <b>SIO_FLUSH</b> IOCTL command.
<b>WSAEFAULT</b>	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if the <i>lpvInBuffer</i> , <i>lpvoutBuffer</i> , <i>lpcbBytesReturned</i> , <i>lpOverlapped</i> or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.
<b>WSAEINPROGRESS</b>	A blocking operation is currently executing. This error is returned if the function is invoked when a callback is in progress.
<b>WSAEINTR</b>	A blocking operation was interrupted by a call to <b>WSACancelBlockingCall</b> . This error is returned if a blocking operation was interrupted.
<b>WSAEINVAL</b>	An invalid argument was supplied. This error is returned if the <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified.
<b>WSAENETDOWN</b>	A socket operation encountered a dead network. This error is returned if the network subsystem has failed.
<b>WSAENOTSOCK</b>	An operation was attempted on something that is not a socket. This error is returned if the descriptor <i>s</i> is not a socket.
<b>WSAEOPNOTSUPP</b>	The attempted operation is not supported for the type of object referenced. This error is returned if the specified IOCTL command is not supported. This error is also returned if the <b>SIO_ACQUIRE_PORT_RESERVATION</b> IOCTL is not supported by the transport provider. This error is also returned when an attempt to use the <b>SIO_ACQUIRE_PORT_RESERVATION</b> IOCTL is made on a socket other than UDP or TCP.

## Remarks

The **SIO\_ACQUIRE\_PORT\_RESERVATION** IOCTL is supported on Windows Vista and later versions of the operating system.

Applications and services which need to reserve ports fall into two categories. The first category includes components which need a particular port as part of their operation. Such components will generally prefer to specify their required port at installation time (in an application manifest, for example). The second category includes components which need any available port or block of ports at runtime. These two categories correspond to specific and wildcard port reservation requests. Specific reservation requests may be persistent or runtime, while wildcard port reservation requests are only supported at runtime.

The **SIO\_ACQUIRE\_PORT\_RESERVATION** IOCTL is used to request a runtime reservation for a block of TCP or UDP ports. For runtime port reservations, the port pool requires that reservations be consumed from the process on whose socket the reservation was granted. Runtime port reservations last only as long as the lifetime of the socket on which the **SIO\_ACQUIRE\_PORT\_RESERVATION** IOCTL was called. In contrast, persistent port reservations created using the [CreatePersistentTcpPortReservation](#) or [CreatePersistentUdpPortReservation](#) function may be consumed by any process with the ability to obtain persistent reservations.

Once a runtime TCP or UDP port reservation has been obtained, an application can request port assignments from the port reservation by opening a TCP or UDP socket, then calling the [WSAOctl](#) function specifying the **SIO\_ASSOCIATE\_PORT\_RESERVATION** IOCTL and passing the reservation token before issuing a call to the bind function on the socket.

If both *lpOverlapped* and *lpCompletionRoutine* parameters 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, except that the function can block if socket *s* is in blocking mode. If socket *s* is in non-blocking mode, this function will still block since this particular IOCTL does not support non-blocking mode.

For overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time.

Any IOCTL may block indefinitely, depending on the service provider's implementation. If the application cannot tolerate blocking in a [WSAOctl](#) or [WSPIoctl](#) function call, overlapped I/O would be advised for IOCTLs that are especially likely to block.

The **SIO\_ACQUIRE\_PORT\_RESERVATION** IOCTL can fail with **WSAEINTR** or **WSA\_OPERATION\_ABORTED** under the following cases:

- The request is canceled by the I/O Manager.
- The socket is closed.

## Examples

The following example acquires a runtime port reservation, then creates a socket and allocates a port from the runtime port reservation for the socket, and then closes the socket and the releases the runtime port reservation.

```
#ifndef UNICODE
#define UNICODE
#endif

#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

#include <Windows.h>
#include <winsock2.h>
#include <mstcpip.h>
#include <ws2ipdef.h>
#include <stdio.h>
#include <stdlib.h>

// Need to link with Ws2_32.lib for Winsock functions
#pragma comment(lib, "ws2_32.lib")

int wmain(int argc, WCHAR ** argv)
{
    // Declare and initialize variables
```

```

int startPort = 0;           // host byte order
int numPorts = 0;
USHORT startPortns = 0;      // Network byte order

INET_PORT_RANGE portRange = { 0 };
INET_PORT_RESERVATION_INSTANCE portRes = { 0 };

unsigned long status = 0;

WSADATA wsaData = { 0 };
int iResult = 0;

SOCKET sock = INVALID_SOCKET;
int iFamily = AF_INET;
int iType = 0;
int iProtocol = 0;

SOCKET sockRes = INVALID_SOCKET;

DWORD bytesReturned = 0;

// Note that the sockaddr_in struct works only with AF_INET not AF_INET6
// An application needs to use the sockaddr_in6 for AF_INET6
sockaddr_in service;
sockaddr_in sockName;
int nameLen = sizeof (sockName);

// Validate the parameters
if (argc != 6) {
    wprintf
        (L"usage: %s <addressfamily> <type> <protocol> <StartingPort> <NumberOfPorts>\n",
        argv[0]);
    wprintf(L"Opens a socket for the specified family, type, & protocol\n");
    wprintf
        (L"and then acquires a runtime port reservation for the protocol specified\n");
    wprintf(L"%ws example usage\n", argv[0]);
    wprintf(L" %ws 2 2 17 5000 20\n", argv[0]);
    wprintf(L" where AF_INET=2 SOCK_DGRAM=2 IPPROTO_UDP=17 StartPort=5000 NumPorts=20", argv[0]);

    return 1;
}

iFamily = _wtoi(argv[1]);
iType = _wtoi(argv[2]);
iProtocol = _wtoi(argv[3]);

startPort = _wtoi(argv[4]);
if (startPort < 0 || startPort > 65535) {
    wprintf(L"Starting point must be either 0 or between 1 and 65,535\n");
    return 1;
}
startPortns = htons((USHORT) startPort);

numPorts = _wtoi(argv[5]);
if (numPorts < 0) {
    wprintf(L"Number of ports must be a positive number\n");
    return 1;
}

portRange.StartPort = startPortns;
portRange.NumberOfPorts = (USHORT) numPorts;

// Initialize Winsock
iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
if (iResult != 0) {
    wprintf(L"WSAStartup failed with error = %d\n", iResult);
    return 1;
}

```

```

sock = socket(iFamily, iType, iProtocol);
if (sock == INVALID_SOCKET) {
    wprintf(L"socket function failed with error = %d\n", WSAGetLastError());
    WSACleanup();
    return 1;
} else {
    wprintf(L"socket function succeeded\n");

    iResult =
        WSAIoctl(sock, SIO_ACQUIRE_PORT_RESERVATION, (LPVOID) & portRange,
                  sizeof (INET_PORT_RANGE), (LPVOID) & portRes,
                  sizeof (INET_PORT_RESERVATION_INSTANCE), &bytesReturned, NULL, NULL);
    if (iResult != 0) {
        wprintf(L"WSAIoctl(SIO_ACQUIRE_PORT_RESERVATION) failed with error = %d\n",
                WSAGetLastError());
        closesocket(sock);
        WSACleanup();
        return 1;
    } else {
        wprintf
            (L"WSAIoctl(SIO_ACQUIRE_PORT_RESERVATION) succeeded, bytesReturned = %u\n",
             bytesReturned);
        wprintf(L" Starting port=%d, Number of Ports=%d, Token=%I64d\n",
                htons(portRes.Reservation.StartPort),
                portRes.Reservation.NumberOfPorts, portRes.Token);

        sockRes = socket(iFamily, iType, iProtocol);
        if (sockRes == INVALID_SOCKET) {
            wprintf(L"socket function for second socket failed with error = %d\n",
                    WSAGetLastError());
            closesocket(sock);
            WSACleanup();
            return 1;
        } else {
            wprintf(L"socket function for second socket succeeded\n");

            iResult =
                WSAIoctl(sock, SIO_ASSOCIATE_PORT_RESERVATION,
                          (LPVOID) & portRes.Token, sizeof (ULONG64), NULL, 0,
                          &bytesReturned, NULL, NULL);
            if (iResult != 0) {
                wprintf
                    (L"WSAIoctl(SIO_ASSOCIATE_PORT_RESERVATION) failed with error = %d\n",
                     WSAGetLastError());
            } else {
                wprintf
                    (L"WSAIoctl(SIO_ASSOCIATE_PORT_RESERVATION) succeeded, bytesReturned = %u\n",
                     bytesReturned);

                service.sin_family = (ADDRESS_FAMILY) iFamily;
                service.sin_addr.s_addr = INADDR_ANY;
                service.sin_port = 0;

                iResult = bind(sock, (SOCKADDR *) & service, sizeof (service));
                if (iResult == SOCKET_ERROR)
                    wprintf(L"bind failed with error = %d\n", WSAGetLastError());
                else {
                    wprintf(L"bind succeeded\n");
                    iResult = getsockname(sock, (SOCKADDR *) & sockName, &nameLen);
                    if (iResult == SOCKET_ERROR)
                        wprintf(L"getsockname failed with error = %d\n",
                               WSAGetLastError());
                    else {
                        wprintf(L"getsockname succeeded\n");
                        wprintf(L"Port number allocated = %u\n",
                               ntohs(sockName.sin_port));
                    }
                }
            }
        }
    }
}

```

```

    }

    // comment out this block of code if you don't want to delete the reservation just created
    iResult =
        WSAIoctl(sock, SIO_RELEASE_PORT_RESERVATION, (LPVOID) & portRes.Token,
                  sizeof (ULONG64), NULL, 0, &bytesReturned, NULL, NULL);
    if (iResult != 0) {
        wprintf(
            L"WSAIoctl(SIO_RELEASE_PORT_RESERVATION) failed with error = %d\n",
            WSAGetLastError());
    } else {
        wprintf(
            L"WSAIoctl(SIO_RELEASE_PORT_RESERVATION) succeeded, bytesReturned = %u\n",
            bytesReturned);
    }
}

if (sockRes != INVALID_SOCKET) {
    iResult = closesocket(sockRes);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"closesocket for second socket failed with error = %d\n",
                WSAGetLastError());
    }
}
if (sock != INVALID_SOCKET) {
    iResult = closesocket(sock);
    if (iResult == SOCKET_ERROR) {
        wprintf(L"closesocket for first socket failed with error = %d\n",
                WSAGetLastError());
    }
}
}

WSACleanup();

return 0;
}

```

## See also

[accept](#)

[bind](#)

[CreatePersistentTcpPortReservation](#)

[CreatePersistentUdpPortReservation](#)

[DeletePersistentTcpPortReservation](#)

[DeletePersistentUdpPortReservation](#)

[INET\\_PORT\\_RANGE](#)

[INET\\_PORT\\_RESERVATION\\_INSTANCE](#)

[LookupPersistentTcpPortReservation](#)

[LookupPersistentUdpPortReservation](#)

[SIO\\_ASSOCIATE\\_PORT\\_RESERVATION](#)

[SIO\\_RELEASE\\_PORT\\_RESERVATION](#)

[socket](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_ADDRESS\_LIST\_QUERY Control Code

3/5/2021 • 5 minutes to read • [Edit Online](#)

## Description

The SIO\_ADDRESS\_LIST\_QUERY control code obtains a list of local transport addresses of the socket's protocol family to which the application can bind. The list of addresses varies based on address family and some addresses are excluded from the list.

To perform this operation, call the [WSAIoctl](#) or [WSPIoctl](#) function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_ADDRESS_LIST_QUERY, // dwIoControlCode
    NULL,                // lpvInBuffer
    0,                   // cbInBuffer
    (LPVOID) lpvOutBuffer, // output buffer
    (DWORD) cbOutBuffer,  // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_ADDRESS_LIST_QUERY, // dwIoControlCode
    NULL,                // lpvInBuffer
    0,                   // cbInBuffer
    (LPVOID) lpvOutBuffer, // output buffer
    (DWORD) cbOutBuffer,  // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);
```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use SIO\_ADDRESS\_LIST\_QUERY for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter is unused for this operation.

### cbInBuffer

The size, in bytes, of the input buffer. This parameter is unused for this operation.

### lpvOutBuffer

A pointer to the output buffer.

### **cbOutBuffer**

The size, in bytes, of the output buffer.

### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer.

### **lpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAOctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

### **lpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

### **lpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#).

The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

### **lpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAOctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAOctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
<b>WSA_IO_PENDING</b>	An overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSA_OPERATION_ABORTED</b>	An overlapped operation was canceled due to the closure of the socket or the execution of the <b>SIO_FLUSH</b> IOCTL command.
<b>WSAEFAULT</b>	The <i>lpOverlapped</i> or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.
<b>WSAEINPROGRESS</b>	The function is invoked when a callback is in progress.

ERROR CODE	MEANING
WSAEINTR	A blocking operation was interrupted.
WSAEINVAL	The <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified. This error is returned if the <i>cbInBuffer</i> parameter is not set to <b>NULL</b> .
WSAENETDOWN	The network subsystem has failed.
WSAENOBUFS	No buffer space available.
WSAENOPROTOOPT	The socket option is not supported on the specified protocol.
WSAENOTSOCK	The descriptor <i>s</i> is not a socket.
WSAEOPNOTSUPP	The specified IOCTL command is not supported. This error is returned if the <b>SIO_ADDRESS_LIST_QUERY</b> IOCTL is not supported by the transport provider.

## Remarks

The **SIO\_ADDRESS\_LIST\_QUERY** IOCTL is supported on Windows 2000 and later versions of the operating system.

The **SIO\_ADDRESS\_LIST\_QUERY** control code obtains a list of local transport addresses of the socket's protocol family to which the application can bind. The list of addresses varies based on address family.

For the **AF\_INET6** address family, all addresses are returned except for the following:

- Any IP address where the duplicate address detection (DAD) state is not **IpDadStatePreferred**.
- Any IP address with a scope level lower than **ScopeLevelSubnet** on an interface where the interface type is **IF\_TYPE\_SOFTWARE\_LOOPBACK**. This means link-local (`fe80:*`) and loopback (`::1`) addresses on interfaces of **IF\_TYPE\_SOFTWARE\_LOOPBACK** type are excluded, but not if these addresses are on an interface with a different type.

For the **AF\_INET** address family, all addresses are returned except for the following:

- Any IP address where the duplicate address detection (DAD) state is not **IpDadStatePreferred**.
- Any IP address on an interface where the interface type is **IF\_TYPE\_SOFTWARE\_LOOPBACK** and link is local. This means link-local (`169.254.*`) and loopback (`127`) addresses on interfaces of **IF\_TYPE\_SOFTWARE\_LOOPBACK** type are excluded, but not if these addresses are on an interface with a different type.

For more information on DAD state, see the IP Helper documentation on the **IP\_DAD\_STATE** enumeration and **IP\_ADAPTER\_UNICAST\_ADDRESS** structure and the MIB documentation on the **MIB\_UNICASTIPADDRESS\_ROW** structure. For more information on interface type, see the IP Helper documentation on the **IP\_ADAPTER\_ADDRESSES** structure and the **GetAdaptersAddresses** function and the MIB documentation on **MIB\_IF\_ROW2** structure. For more information on the scope level, see the IP Helper documentation on the **IP\_ADAPTER\_ADDRESSES** structure and the **SCOPE\_LEVEL** enumeration.

The list returned in the output buffer pointed to by the *lpvOutBuffer* parameter is in the form of a **SOCKET\_ADDRESS\_LIST** structure.

If the output buffer specified in the *IpvOutBuffer* parameter 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, in bytes, for the output buffer is returned in the *IpcbBytesReturned* parameter in this case. Note the **WSAEFAULT** error code is also returned if the *IpvInBuffer*, *IpvOutBuffer*, or *IpcbBytesReturned* parameter is not completely contained in a valid part of the user address space.

The **SIO\_ADDRESS\_LIST\_QUERY** IOCTL is normally called synchronously with the *IpvOverlapped* parameter set to **NULL**, since the list of addresses is returned immediately.

**Note** In Windows Plug-n-Play environments, addresses can be added and 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 through the **SIO\_ADDRESS\_LIST\_CHANGE** IOCTL which provides for notification through either overlapped I/O 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 the **SIO\_ADDRESS\_LIST\_CHANGE** IOCTL
- Issue the **SIO\_ADDRESS\_LIST\_QUERY** IOCTL
- Whenever the **SIO\_ADDRESS\_LIST\_CHANGE** IOCTL call notifies the application of an address list change (either through overlapped I/O or by signaling **FD\_ADDRESS\_LIST\_CHANGE** event), the whole sequence of actions should be repeated.

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, the organization of header files has changed and the **SIO\_ADDRESS\_LIST\_QUERY** control code is defined in the *Ws2def.h* header file. Note that the *Ws2def.h* header file is automatically included in *Winsock2.h*, and should never be used directly.

## See also

[GetAdaptersAddresses](#)

[IP\\_ADAPTER\\_ADDRESSES](#)

[IP\\_ADAPTER\\_UNICAST\\_ADDRESS](#)

[IP\\_DAD\\_STATE](#)

[MIB\\_IF\\_ROW2](#)

[MIB\\_UNICASTIPADDRESS\\_ROW](#)

[SCOPE\\_LEVEL](#)

[SOCKET\\_ADDRESS\\_LIST](#)

[socket](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_APPLY\_TRANSPORT\_SETTING Control Code

3/5/2021 • 5 minutes to read • [Edit Online](#)

## Description

The **SIO\_APPLY\_TRANSPORT\_SETTING** control code applies one or more transport settings to a socket.

To perform this operation, call the **WSAIoctl** or **WSPIoctl** function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_APPLY_TRANSPORT_SETTING, // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to the input buffer
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    (LPVOID) lpvOutBuffer, // pointer to the output buffer
    (DWORD) cbOutBuffer,  // size, in bytes, of the output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_APPLY_TRANSPORT_SETTING, // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to the input buffer
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    (LPVOID) lpvOutBuffer, // pointer to the output buffer
    (DWORD) cbOutBuffer,  // size, in bytes, of the output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);
```

## Parameters

**s**

A descriptor identifying a socket.

**dwIoControlCode**

The control code for the operation. Use **SIO\_APPLY\_TRANSPORT\_SETTING** for this operation.

**lpvInBuffer**

A pointer to the input buffer. This parameter contains a pointer to a structure where the first member of the structure is a **TRANSPORT\_SETTING\_ID** structure that determines what transport setting is being applied.

**cbInBuffer**

The size, in bytes, of the input buffer. This parameter depends on the transport setting being applied.

**lpvOutBuffer**

A pointer to the output buffer. This parameter depends on the transport setting being applied.

**cbOutBuffer**

The size, in bytes, of the output buffer.

### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer.

If the output buffer is too small, the call fails, [WSAGetLastError](#) returns [WSAEINVAL](#), and the *lpcbBytesReturned* parameter points to a **DWORD** value of zero.

If *lpOverlapped* is **NULL**, the **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned on a successful call cannot be zero.

If the *lpOverlapped* parameter is not **NULL** for overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned may be zero since the size of the data stored can't be determined until the overlapped operation has completed. The final completion status can be retrieved when the appropriate completion method is signaled when the operation has completed.

### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAIoctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

### **lpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

### **lpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

### **lpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAIoctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAIoctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
<b>WSA_IO_PENDING</b>	Overlapped I/O operation is in progress. This value is returned if an overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSA_OPERATION_ABORTED</b>	The I/O operation has been aborted because of either a thread exit or an application request. This error is returned if an overlapped operation was canceled due to the closure of the socket or the execution of the <b>SIO_FLUSH</b> IOCTL command.
<b>WSAEFAULT</b>	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if the <i>lpvInBuffer</i> , <i>lpvoutBuffer</i> , <i>lpcbBytesReturned</i> , <i>lpOverlapped</i> or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.
<b>WSAEINPROGRESS</b>	A blocking operation is currently executing. This error is returned if the function is invoked when a callback is in progress.
<b>WSAEINTR</b>	A blocking operation was interrupted by a call to <b>WSACancelBlockingCall</b> . This error is returned if a blocking operation was interrupted.
<b>WSAEINVAL</b>	An invalid argument was supplied. This error is returned if the <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified.
<b>WSAENETDOWN</b>	A socket operation encountered a dead network. This error is returned if the network subsystem has failed.
<b>WSAENOTSOCK</b>	An operation was attempted on something that is not a socket. This error is returned if the descriptor <i>s</i> is not a socket.
<b>WSAEOPNOTSUPP</b>	The attempted operation is not supported for the type of object referenced. This error is returned if the specified IOCTL command is not supported. This error is also returned if the <b>SIO_APPLY_TRANSPORT_SETTING</b> IOCTL is not supported by the transport provider. This error is also returned when an attempt to use the <b>SIO_APPLY_TRANSPORT_SETTING</b> IOCTL is made on a socket other than UDP or TCP.

## Remarks

The **SIO\_APPLY\_TRANSPORT\_SETTING** IOCTL is supported on Windows 8, and Windows Server 2012, and later versions of the operating system.

The **SIO\_APPLY\_TRANSPORT\_SETTING** IOCTL is a generic IOCTL used to apply transport setting to socket. The transport setting being applied is based on the **TRANSPORT\_SETTING\_ID** passed in the *lpvInBuffer* parameter.

Starting with Windows 8 and Windows Server 2012, the system defines the **REAL\_TIME\_NOTIFICATION\_CAPABILITY** capability on a TCP socket. Starting with Windows 10 and Windows

Server 2016, `ASSOCIATE_NAMERES_CONTEXT` is also defined. For more information, see [addrinfoex4](#) and [ASSOCIATE\\_NAMERES\\_CONTEXT\\_INPUT](#).

If the `TRANSPORT_SETTING_ID` passed in the `IpvInBuffer` parameter has the `Guid` member set to `REAL_TIME_NOTIFICATION_CAPABILITY`, then this is a request to apply real time notification settings for the TCP socket used with the [ControlChannelTrigger](#) to receive background network notifications in a Windows Store app. The `IpvInBuffer` parameter should point to a `REAL_TIME_NOTIFICATION_SETTING_INPUT` structure. The `IpvOutBuffer` parameter is unused for this operation. This transport setting applies only to TCP sockets. This transport setting provides a way for WinInet or similar network services to mark a given TCP socket as [ControlChannelTrigger](#) enabled. Windows will mark the corresponding TCP socket and configure appropriate hardware and software settings when this option is called. A Windows Store app will not call this IOCTL directly.

## See also

[ControlChannelTrigger](#)

[socket](#)

[SIO\\_QUERY\\_TRANSPORT\\_SETTING](#)

[TRANSPORT\\_SETTING\\_ID](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_ASSOCIATE\_PORT\_RESERVATION Control Code

3/5/2021 • 8 minutes to read • [Edit Online](#)

## Description

The **SIO\_ASSOCIATE\_PORT\_RESERVATION** control code associates a socket with a persistent or runtime reservation for a block of TCP or UDP identified by the port reservation token. This IOCTL must be issued before the socket is bound. If and when the socket is bound, the port assigned to it will be selected from the port reservation identified by the given token. If no ports are available from the specified reservation, the **bind** function call will fail.

To perform this operation, call the **WSAIoctl** or **WSPIoctl** function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_ASSOCIATE_PORT_RESERVATION, // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to an INET_PORT_RESERVATION_TOKEN
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    NULL,                // lpvOutBuffer is a pointer to the output buffer
    0,                   // cbOutBuffer is the size, in bytes, of the output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_ASSOCIATE_PORT_RESERVATION, // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to an INET_PORT_RESERVATION_TOKEN
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    NULL,                // lpvOutBuffer is a pointer to the output buffer
    0,                   // cbOutBuffer is the size, in bytes, of the output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);
```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use **SIO\_ASSOCIATE\_PORT\_RESERVATION** for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter contains a pointer to an **INET\_PORT\_RESERVATION\_TOKEN** structure with the token for the TCP or UDP port reservation to associate with the socket.

### cbInBuffer

The size, in bytes, of the input buffer. This parameter must be at least the size of the [INET\\_PORT\\_RESERVATION\\_TOKEN](#) structure.

#### **IpvOutBuffer**

A pointer to the output buffer. This parameter is unused for this operation.

#### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter must be set to zero.

#### **IpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer.

If the output buffer is too small, the call fails, [WSAGetLastError](#) returns [WSAEINVAL](#), and the *IpcbBytesReturned* parameter points to a **DWORD** value of zero.

If *lpOverlapped* is **NULL**, the **DWORD** value pointed to by the *IpcbBytesReturned* parameter that is returned on a successful call cannot be zero.

If the *lpOverlapped* parameter is not **NULL** for overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The **DWORD** value pointed to by the *IpcbBytesReturned* parameter that is returned may be zero since the size of the data stored can't be determined until the overlapped operation has completed. The final completion status can be retrieved when the appropriate completion method is signaled when the operation has completed.

#### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAOctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

#### **IpCompletionRoutine**

Type: \_In\_opt\_ [LPWSAOVERLAPPED\\_COMPLETION\\_ROUTINE](#)

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

#### **IpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

#### **IpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAOctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAIoctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
WSA_IO_PENDING	Overlapped I/O operation is in progress. This value is returned if an overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	The I/O operation has been aborted because of either a thread exit or an application request. This error is returned if an overlapped operation was canceled due to the closure of the socket or the execution of the <b>SIO_FLUSH IOCTL</b> command.
WSAEACCES	An attempt was made to access a socket in a way forbidden by its access permissions. This error is returned under several conditions for persistent port reservations that include the following: the user lacks the required administrative privileges on the local computer or the application is not running in an enhanced shell as the built-in Administrator ( <code>RunAs administrator</code> ).
WSAEFAULT	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if the <i>lpvInBuffer</i> , <i>lpvoutBuffer</i> , <i>lpcbBytesReturned</i> , <i>lpOverlapped</i> or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.
WSAEINPROGRESS	A blocking operation is currently executing. This error is returned if the function is invoked when a callback is in progress.
WSAEINTR	A blocking operation was interrupted by a call to <a href="#">WSACancelBlockingCall</a> . This error is returned if a blocking operation was interrupted.
WSAEINVAL	An invalid argument was supplied. This error is returned if the <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified.
WSAENETDOWN	A socket operation encountered a dead network. This error is returned if the network subsystem has failed.
WSAENOTSOCK	An operation was attempted on something that is not a socket. This error is returned if the descriptor <i>s</i> is not a socket.
WSAEOPNOTSUPP	The attempted operation is not supported for the type of object referenced. This error is returned if the specified IOCTL command is not supported. This error is also returned if the <b>SIO_ASSOCIATE_PORT_RESERVATION IOCTL</b> is not supported by the transport provider. This error is also returned when an attempt to use the <b>SIO_ASSOCIATE_PORT_RESERVATION IOCTL</b> is made on a socket other than UDP or TCP.

## Remarks

The **SIO\_ASSOCIATE\_PORT\_RESERVATION** IOCTL is supported on Windows Vista and later versions of the operating system.

Applications and services which need to reserve ports fall into two categories. The first category includes components which need a particular port as part of their operation. Such components will generally prefer to specify their required port at installation time (in an application manifest, for example). The second category includes components which need any available port or block of ports at runtime. These two categories correspond to specific and wildcard port reservation requests. Specific reservation requests may be persistent or runtime, while wildcard port reservation requests are only supported at runtime.

The **SIO\_ASSOCIATE\_PORT\_RESERVATION** IOCTL is used to associate a TCP or UDP port reservation with either a persistent or runtime reservation.

The [CreatePersistentTcpPortReservation](#) or [CreatePersistentUdpPortReservation](#) function provides the ability for an application or service to reserve a persistent block of TCP or UDP ports. Persistent port reservations are recorded in a persistent store for the TCP or UDP module in Windows. Note that the token for a given persistent port reservation may change each time the system is restarted.

Once a persistent TCP or UDP port reservation has been obtained, an application can request port assignments from the port reservation by opening a TCP or UDP socket, then calling the [WSAIoctl](#) function specifying the **SIO\_ASSOCIATE\_PORT\_RESERVATION** IOCTL and passing the reservation token before issuing a call to the [bind](#) function on the socket.

The **SIO\_ACQUIRE\_PORT\_RESERVATION** IOCTL can be used to request a runtime reservation for a block of TCP or UDP ports. For runtime port reservations, the port pool requires that reservations be consumed from the process on whose socket the reservation was granted. Runtime port reservations last only as long as the lifetime of the socket on which the **SIO\_ACQUIRE\_PORT\_RESERVATION** IOCTL was called. In contrast, persistent port reservations created using the [CreatePersistentTcpPortReservation](#) or [CreatePersistentUdpPortReservation](#) function may be consumed by any process with the ability to obtain persistent reservations.

Once a runtime TCP or UDP port reservation has been obtained, an application can request port assignments from the port reservation by opening a TCP or UDP socket, then calling the [WSAIoctl](#) function specifying the **SIO\_ASSOCIATE\_PORT\_RESERVATION** IOCTL and passing the reservation token before issuing a call to the [bind](#) function on the socket.

If both *lpOverlapped* and *lpCompletionRoutine* parameters 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, except that the function can block if socket *s* is in blocking mode. If socket *s* is in non-blocking mode, this function will still block since this particular IOCTL does not support non-blocking mode.

For overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time.

Any IOCTL may block indefinitely, depending on the service provider's implementation. If the application cannot tolerate blocking in a [WSAIoctl](#) or [WSPIoctl](#) function call, overlapped I/O would be advised for IOCTLs that are especially likely to block.

The **SIO\_ASSOCIATE\_PORT\_RESERVATION** IOCTL can fail with **WSAEINTR** or **WSA\_OPERATION\_ABORTED** under the following cases:

- The request is canceled by the I/O Manager.
- The socket is closed.

The **SIO\_ASSOCIATE\_PORT\_RESERVATION** IOCTL passed to the [WSAIoctl](#) or [WSPIoctl](#) function for a

persistent port reservation can only be used in an application when the user is logged on as a member of the Administrators group. If **SIO\_ASSOCIATE\_PORT\_RESERVATION** IOCTL is used in an application when the user is not a member of the Administrators group, the function call will fail and **WSAEACCES** is returned. The use of the **SIO\_ASSOCIATE\_PORT\_RESERVATION** IOCTL can also fail because of user account control (UAC) on Windows Vista and later. If an application that uses this IOCTL with a persistent port reservation is executed by a user logged on as a member of the Administrators group other than the built-in Administrator, this call will fail unless the application has been marked in the manifest file with a **requestedExecutionLevel** set to *requireAdministrator*. If the application lacks this manifest file, a user logged on as a member of the Administrators group other than the built-in Administrator must then be executing the application in an enhanced shell as the built-in Administrator (`RunAs administrator`) for this function to succeed.

## See also

[bind](#)

[CreatePersistentTcpPortReservation](#)

[CreatePersistentUdpPortReservation](#)

[DeletePersistentTcpPortReservation](#)

[DeletePersistentUdpPortReservation](#)

[INET\\_PORT\\_RESERVATION\\_TOKEN](#)

[LookupPersistentTcpPortReservation](#)

[LookupPersistentUdpPortReservation](#)

[SIO\\_ACQUIRE\\_PORT\\_RESERVATION](#)

[SIO\\_RELEASE\\_PORT\\_RESERVATION](#)

[socket](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_IDEAL\_SEND\_BACKLOG\_CHANGE Control Code

3/5/2021 • 6 minutes to read • [Edit Online](#)

## Description

The SIO\_IDEAL\_SEND\_BACKLOG\_CHANGE control code notifies an application when the ideal send backlog (ISB) value changes for the connection.

To perform this operation, call the **WSAIoctl** or **WSPIoctl** function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_IDEAL_SEND_BACKLOG_CHANGE, // dwIoControlCode
    NULL,                // lpvInBuffer
    0,                  // cbInBuffer
    NULL,                // output buffer
    0,                  // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);

```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_IDEAL_SEND_BACKLOG_CHANGE, // dwIoControlCode
    NULL,                // lpvInBuffer
    0,                  // cbInBuffer
    NULL,                // output buffer
    0,                  // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);

```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use SIO\_IDEAL\_SEND\_BACKLOG\_CHANGE for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter is unused for this operation.

### cbInBuffer

The size, in bytes, of the input buffer. This parameter is unused for this operation.

### lpvOutBuffer

A pointer to the output buffer. This parameter is unused for this operation.

### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter must be set to zero.

### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer. This returned parameter points to a **DWORD** value of zero for this operation, since there is no output.

### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAIoctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

### **lpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

### **lpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

### **lpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAIoctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAIoctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
<b>WSA_IO_PENDING</b>	An overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSA_OPERATION_ABORTED</b>	An overlapped operation was canceled due to the closure of the socket or the execution of the <b>SIO_FLUSH</b> IOCTL command.
<b>WSAEFAULT</b>	The <i>lpOverlapped</i> or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.

ERROR CODE	MEANING
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAEINTR	A blocking operation was interrupted.
WSAEINVAL	The <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified. This error is returned if the <i>cbOutBuffer</i> parameter is not zero.
WSAENETDOWN	The network subsystem has failed.
WSAENOPROTOOPT	The socket option is not supported on the specified protocol.
WSAENOTCONN	The socket <i>s</i> is not connected.
WSAENOTSOCK	The descriptor <i>s</i> is not a socket.
WSAEOPNOTSUPP	The specified IOCTL command is not supported. This error is returned if the SIO_IDEAL_SEND_BACKLOG_CHANGE IOCTL is not supported by the transport provider. This error is also returned when an attempt to use the SIO_IDEAL_SEND_BACKLOG_CHANGE IOCTL is made on a datagram socket.

## Remarks

The SIO\_IDEAL\_SEND\_BACKLOG\_CHANGE IOCTL is supported on Windows Server 2008, Windows Vista with Service Pack 1 (SP1), and later versions of the operating system.

When sending data over a TCP connection using Windows sockets, it is important to keep a sufficient amount of data outstanding (sent but not acknowledged yet) in TCP in order to achieve the highest throughput. The ideal value for the amount of data outstanding to achieve the best throughput for the TCP connection is called the ideal send backlog (ISB) size. The ISB value is a function of the bandwidth-delay product of the TCP connection and the receiver's advertised receive window (and partly the amount of congestion in the network).

The ISB value per connection is available from the TCP protocol implementation in Windows Server 2008, Windows Vista with SP1, and later versions of the operating system. The SIO\_IDEAL\_SEND\_BACKLOG\_CHANGE IOCTL can be used by an application to get a notification when the ISB value changes dynamically for a connection.

On Windows Server 2008, Windows Vista with SP1, and later versions of the operating system, the SIO\_IDEAL\_SEND\_BACKLOG\_CHANGE and SIO\_IDEAL\_SEND\_BACKLOG\_QUERY IOCTLs are supported on stream-oriented sockets that are in a connected state.

The range for the ISB value for a TCP connection can theoretically vary from 0 to a maximum of 16 megabytes.

See the [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#) IOCTL reference page for typical usage of the ISB mechanism for achieving better throughput over high bandwidth-delay product connections.

The SIO\_IDEAL\_SEND\_BACKLOG\_CHANGE IOCTL is allowed only on a stream socket that is in the connected state. Otherwise the [WSAIoctl](#) or [WSPIoctl](#) function will fail with WSAENOTCONN.

The SIO\_IDEAL\_SEND\_BACKLOG\_CHANGE IOCTL uses no input or output buffers and pends or blocks until

an ISB change occurs on the underlying connection. When this IOCTL is completed, a Winsock application can use the [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#) IOCTL to retrieve the new ISB value on the connection.

The [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#) IOCTL does not support non-blocking mode. An application is allowed to issue this IOCTL on a non-blocking socket, but the IOCTL will simply block or pend until the ISB value changes.

If both *lpOverlapped* and *lpCompletionRoutine* parameters 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, except that the function can block if socket *s* is in blocking mode. If socket *s* is in non-blocking mode, this function will still block since this particular IOCTL does not support non-blocking mode.

For overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time.

Any IOCTL may block indefinitely, depending on the service provider's implementation. If the application cannot tolerate blocking in a [WSAIoctl](#) or [WSPIoctl](#) function call, overlapped I/O would be advised for IOCTLs that are especially likely to block.

The [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#) IOCTL provides a notification and is expected to block or pend until the ISB value changes. So it would commonly be called asynchronously with the *lpOverlapped* parameter set to a valid WSAOVERLAPPED structure.

The [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#) IOCTL can fail with [WSAEINTR](#) or [WSA\\_OPERATION\\_ABORTED](#) under the following cases:

- The TCP connection is gracefully disconnected in the send direction. This can occur as a result of a call to shutdown function with the how parameter set to SD\_SEND, a call to the [DisconnectEx](#) function, or a call to the [TransmitFile](#) or [TransmitPackets](#) function with *dwFlags* parameter set to TF\_DISCONNECT or TF\_REUSE.
- The TCP connection has been reset or aborted.
- The application issues a [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#) IOCTL when there is already a pended notification request. Only one pending [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#) request is allowed at a time.
- The request is canceled by the I/O Manager.
- The socket is closed.

Two inline wrapper functions for these IOCTLs are defined in the *Ws2tcpip.h* header file. It is recommended that these inline functions be used instead of using the [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#) and [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#) IOCTLs directly.

The inline wrapper function for the [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#) IOCTL is the [idealsendbacklognotify](#) function.

The inline wrapper function for the [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#) IOCTL is the [idealsendbacklogquery](#) function.

## See also

[SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#)

[socket](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

**WSAOVERLAPPED**

**WSASocketA**

**WSASocketW**

# SIO\_IDEAL\_SEND\_BACKLOG\_QUERY Control Code

3/5/2021 • 8 minutes to read • [Edit Online](#)

## Description

The SIO\_IDEAL\_SEND\_BACKLOG\_QUERY control code retrieves the ideal send backlog (ISB) value for the underlying connection.

To perform this operation, call the [WSAIoctl](#) or [WSPIoctl](#) function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_IDEAL_SEND_BACKLOG_QUERY, // dwIoControlCode
    NULL,                // lpvInBuffer
    0,                   // cbInBuffer
    (LPVOID) lpvOutBuffer, // output buffer
    (DWORD) cbOutBuffer,   // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_IDEAL_SEND_BACKLOG_QUERY, // dwIoControlCode
    NULL,                // lpvInBuffer
    0,                   // cbInBuffer
    (LPVOID) lpvOutBuffer, // output buffer
    (DWORD) cbOutBuffer,   // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);
```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use SIO\_IDEAL\_SEND\_BACKLOG\_QUERY for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter is unused for this operation.

### cbInBuffer

The size, in bytes, of the input buffer. This parameter is unused for this operation.

### lpvOutBuffer

A pointer to the output buffer. This parameter should point to a **ULONG** data type if the *lpOverlapped* and *lpCompletionRoutine* parameters are **NULL**.

### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter must be at least the size of a **ULONG** data type.

### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer.

If the output buffer is too small, the call fails, [WSAGetLastError](#) returns [WSAEINVAL](#), and the *lpcbBytesReturned* parameter points to a **DWORD** value of zero.

If *lpOverlapped* is **NULL**, the **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned on a successful call cannot be zero.

If the *lpOverlapped* parameter is not **NULL** for overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned may be zero since the size of the data stored can't be determined until the overlapped operation has completed. The final completion status can be retrieved when the appropriate completion method is signaled when the operation has completed.

### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAOIoctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

### **lpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

### **lpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

### **lpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAOIoctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAOIoctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
<b>WSA_IO_PENDING</b>	An overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSA_OPERATION_ABORTED</b>	An overlapped operation was canceled due to the closure of the socket or the execution of the <b>SIO_FLUSH</b> IOCTL command.
<b>WSAEFAULT</b>	The <i>lpvInBuffer</i> , <i>lpvoutBuffer</i> , <i>lpcbBytesReturned</i> , <i>lpOverlapped</i> or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.
<b>WSAEINPROGRESS</b>	The function is invoked when a callback is in progress.
<b>WSAEINTR</b>	A blocking operation was interrupted.
<b>WSAEINVAL</b>	The <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified. This error is returned if the <i>cbOutBuffer</i> parameter is less than the size of a <b>ULONG</b> data type.
<b>WSAENETDOWN</b>	The network subsystem has failed.
<b>WSAENOPROTOOPT</b>	The socket option is not supported on the specified protocol.
<b>WSAENOTCONN</b>	The socket <i>s</i> is not connected.
<b>WSAENOTSOCK</b>	The descriptor <i>s</i> is not a socket.
<b>WSAEOPNOTSUPP</b>	The specified IOCTL command is not supported. This error is returned if the <b>SIO_IDEAL_SEND_BACKLOG_QUERY</b> IOCTL is not supported by the transport provider. This error is also returned when an attempt to use the <b>SIO_IDEAL_SEND_BACKLOG_QUERY</b> IOCTL is made on a datagram socket.

## Remarks

The **SIO\_IDEAL\_SEND\_BACKLOG\_QUERY** IOCTL is supported on Windows Server 2008, Windows Vista with Service Pack 1 (SP1), and later versions of the operating system.

When sending data over a TCP connection using Windows sockets, it is important to keep a sufficient amount of data outstanding (sent but not acknowledged yet) in TCP in order to achieve the highest throughput. The ideal value for the amount of data outstanding to achieve the best throughput for the TCP connection is called the ideal send backlog (ISB) size. The ISB value is a function of the bandwidth-delay product of the TCP connection and the receiver's advertised receive window (and partly the amount of congestion in the network).

The ISB value per connection is available from the TCP protocol implementation in Windows Server 2008, Windows Vista with SP1, and later versions of the operating system. The **SIO\_IDEAL\_SEND\_BACKLOG\_QUERY** IOCTL can be used by an application to get a notification when the ISB value changes dynamically for a connection.

On Windows Server 2008, Windows Vista with SP1, and later versions of the operating system, the

[SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#) and [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#) IOCTLs are supported on stream-oriented sockets that are in a connected state.

The range for the ISB value for a TCP connection can theoretically vary from 0 to a maximum of 16 megabytes.

The typical usage of the [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#) and [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#) IOCTLs is based on the send method used by the applications. Two common cases are discussed.

Applications that perform one blocking or non-blocking send request at a time typically rely on internal send buffering by Winsock to achieve decent throughput. The send buffer limit for a given connection is controlled by the [SO\\_SNDBUF](#) socket option. For the blocking and non-blocking send method, the send buffer limit determines how much data is kept outstanding in TCP. If the ISB value for the connection is larger than the send buffer limit, then the throughput achieved on the connection will not be optimal. In order to achieve better throughput, the applications can set the send buffer limit based on the result of the ISB query as ISB change notifications occur on the connection.

For applications that use the overlapped send method with multiple send requests outstanding, the amount of data kept outstanding in TCP is determined by the send buffer limit in Winsock and the total amount of data contained in the outstanding overlapped send requests. In this case, applications should use the ISB value to determine how many outstanding send requests they should keep and what the data size for each send request should be. Ideally, the application should try to keep the following equation satisfied:

```
ISB value == send buffer limit + (number of simultaneous overlapped send requests * data length per send request)
```

Note that using the ISB IOCTLs over TCP sockets in the above fashion can lead to increased memory usage in exchange for increased throughput on connections with a high bandwidth-delay product. The TCP implementation in Windows will throttle ISB values as necessary based on overall system memory usage.

The [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#) IOCTL is allowed only on a stream socket that is in the connected state. Otherwise the [WSAOctl](#) or [WSPIoctl](#) function will fail with [WSAENOTCONN](#).

Any IOCTL may block indefinitely, depending on the service provider's implementation. If the application cannot tolerate blocking in a [WSAOctl](#) or [WSPIoctl](#) function call, overlapped I/O would be advised for IOCTLs that are especially likely to block.

The [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#) IOCTL is not likely to block so it is normally called synchronously with the *lpOverlapped* and *lpCompletionRoutine* parameters set to **NULL**.

The [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#) IOCTL can fail with [WSAEINTR](#) or [WSA\\_OPERATION\\_ABORTED](#) under the following cases:

The TCP connection is gracefully disconnected in the send direction. This can occur as a result of a call to shutdown function with the how parameter set to [SD\\_SEND](#), a call to the [DisconnectEx](#) function, or a call to the [TransmitFile](#) or [TransmitPackets](#) function with *dwFlags* parameter set to [TF\\_DISCONNECT](#) or [TF\\_REUSE](#). The TCP connection has been reset or aborted. The request is canceled by the I/O Manager.

Two inline wrapper functions for these IOCTLs are defined in the *Ws2tcpip.h* header file. It is recommended that these inline functions be used instead of using the [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#) and [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#) IOCTLs directly.

The inline wrapper function for the [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#) IOCTL is the **idealsendbacklognotify** function.

The inline wrapper function for the [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#) IOCTL is the **idealsendbacklogquery** function.

Dynamic send buffering for TCP was added on Windows 7 and Windows Server 2008 R2. By default, dynamic send buffering for TCP is enabled unless an application sets the [SO\\_SNDBUF](#) socket option on the stream

socket.

Using netsh is the recommended method to query or set dynamic send buffering for TCP.

The current value for dynamic send buffering for TCP can be retrieved using the following command:

```
netsh winsock show autotuning
```

Dynamic send buffering for TCP can be disabled using the following command:

```
netsh winsock set autotuning off
```

Dynamic send buffering for TCP can be enabled using the following command:

```
netsh winsock set autotuning on
```

While it is discouraged, dynamic send buffering can be disabled from an application by setting the following registry value to zero:

```
HKEY_LOCAL_MACHINE\SYSTEM\Current Control Set\Services\AFD\Parameters\DynamicSendBufferDisable
```

When changing the value for dynamic send buffering using NetSh.exe or changing the registry value, the computer must be restarted for the change to take effect.

With dynamic send buffering on Windows 7 and Windows Server 2008 R2, the use of the [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#) and [SIO\\_IDEAL\\_SEND\\_BACKLOG\\_QUERY](#) IOCTLs are needed only in special circumstances.

## See also

[SIO\\_IDEAL\\_SEND\\_BACKLOG\\_CHANGE](#)

[socket](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_KEEPALIVE\_VALS Control Code

3/5/2021 • 5 minutes to read • [Edit Online](#)

## Description

The SIO\_KEEPALIVE\_VALS control code enables or disables the per-connection setting of the TCP keep-alive option which specifies the TCP keep-alive timeout and interval.

To perform this operation, call the [WSAIoctl](#) or [WSPIoctl](#) function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_KEEPALIVE_VALS,   // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to tcp_keepalive struct
    (DWORD) cbInBuffer,   // length of input buffer
    NULL,                // output buffer
    0,                   // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_KEEPALIVE_VALS,   // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to tcp_keepalive struct
    (DWORD) cbInBuffer,   // length of input buffer
    NULL,                // output buffer
    0,                   // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);
```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use SIO\_KEEPALIVE\_VALS for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter should point to a `tcp_keepalive` structure.

### cbInBuffer

The size, in bytes, of the input buffer. This parameter should equal to or greater than the size of the `tcp_keepalive` structure pointed to by the `lpvInBuffer` parameter.

### lpvOutBuffer

A pointer to the output buffer. This parameter is unused for this operation.

### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter must be set to zero.

### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer. This returned parameter points to a **DWORD** value of zero for this operation, since there is no output.

### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAIoctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

### **lpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

### **lpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

### **lpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAIoctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAIoctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
<b>WSA_IO_PENDING</b>	An overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSA_OPERATION_ABORTED</b>	An overlapped operation was canceled due to the closure of the socket or the execution of the <b>SIO_FLUSH</b> IOCTL command.
<b>WSAEFAULT</b>	The <i>lpOverlapped</i> or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.

ERROR CODE	MEANING
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAEINTR	A blocking operation was interrupted.
WSAEINVAL	The <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified.
WSAENETDOWN	The network subsystem has failed.
WSAENOPROTOOPT	The socket option is not supported on the specified protocol. This error is returned for a datagram socket.
WSAENOTSOCK	The descriptor s is not a socket.
WSAEOPNOTSUPP	The specified IOCTL command is not supported. This error is returned if the SIO_KEEPALIVE_VALS IOCTL is not supported by the transport provider.

## Remarks

The SIO\_KEEPALIVE\_VALS IOCTL is supported on Windows 2000 and later versions of the operating system.

The SIO\_KEEPALIVE\_VALS control code enables or disables the per-connection setting of the TCP keep-alive option which specifies the TCP keep-alive timeout and interval used for TCP keep-alive packets. For more information on the keep-alive option, see section 4.2.3.6 on the Requirements for Internet Hosts—Communication Layers specified in RFC 1122 available at the [IETF website](#). (This resource may only be available in English.)

The *lpvInBuffer* parameter should point to a **tcp\_keepalive** structure defined in the *Mstcpip.h* header file. This structure is defined as follows:

```
/* Argument structure for SIO_KEEPALIVE_VALS */
struct tcp_keepalive {
    u_long onoff;
    u_long keepalivetime;
    u_long keepaliveinterval;
};
```

The value specified in the **onoff** member determines if TCP keep-alive is enabled or disabled. If the **onoff** member is set to a nonzero value, TCP keep-alive is enabled and the other members in the structure are used. The **keepalivetime** member specifies the timeout, in milliseconds, with no activity until the first keep-alive packet is sent. The **keepaliveinterval** member specifies the interval, in milliseconds, between when successive keep-alive packets are sent if no acknowledgement is received.

The **SO\_KEEPALIVE** option, which is one of the [SOL\\_SOCKET Socket Options](#), can also be used to enable or disable the TCP keep-alive on a connection, as well as query the current state of this option. To query whether TCP keep-alive is enabled on a socket, the **getsockopt** function can be called with the **SO\_KEEPALIVE** option. To enable or disable TCP keep-alive, the **setsockopt** function can be called with the **SO\_KEEPALIVE** option. If TCP keep-alive is enabled with **SO\_KEEPALIVE**, then the default TCP settings are used for keep-alive timeout and interval unless these values have been changed using SIO\_KEEPALIVE\_VALS.

The default system-wide value of the keep-alive timeout is controllable through the [KeepAliveTime](#) registry

setting which takes a value in milliseconds. If the key is not set, the default keep-alive timeout is 2 hours. The default system-wide value of the keep-alive interval is controllable through the [KeepAliveInterval](#) registry setting which takes a value in milliseconds. If the key is not set, the default keep-alive interval is 1 second.

On Windows Vista and later, the number of keep-alive probes (data retransmissions) is set to 10 and cannot be changed.

On Windows Server 2003, Windows XP, and Windows 2000, the default setting for number of keep-alive probes is 5. The number of keep-alive probes is controllable through the [TcpMaxDataRetransmissions](#) and [PPTPTcpMaxDataRetransmissions](#) registry settings. The number of keep-alive probes is set to the larger of the two registry key values. If this number is 0, then keep-alive probes will not be sent. If this number is above 255, then it is adjusted to 255.

## See also

[KeepAliveTime](#)

[KeepAliveInterval](#)

[PPTPTcpMaxDataRetransmissions](#)

[socket](#)

[SO\\_KEEPALIVE](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_LOOPBACK\_FAST\_PATH Control Code

3/5/2021 • 7 minutes to read • [Edit Online](#)

## Description

The SIO\_LOOPBACK\_FAST\_PATH control code configures a TCP socket for lower latency and faster operations on the loopback interface.

**Important** The SIO\_LOOPBACK\_FAST\_PATH is deprecated and is not recommended to be used in your code.

To perform this operation, call the [WSAIoctl](#) or [WSPIoctl](#) function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_LOOPBACK_FAST_PATH, // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to a Boolean value
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    (LPVOID) lpvOutBuffer, // pointer to output buffer
    (DWORD) cbOutBuffer,  // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_LOOPBACK_FAST_PATH, // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to a Boolean value
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    (LPVOID) lpvOutBuffer, // pointer to output buffer
    (DWORD) cbOutBuffer,  // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);
```

## Parameters

**s**

A descriptor identifying a socket.

**dwIoControlCode**

The control code for the operation. Use SIO\_LOOPBACK\_FAST\_PATH for this operation.

**lpvInBuffer**

A pointer to the input buffer. This parameter contains a pointer to a Boolean value that indicates if the socket should be configured for fast loopback operations.

**cbInBuffer**

The size, in bytes, of the input buffer.

**lpvOutBuffer**

A pointer to the output buffer. This parameter is unused for this operation.

#### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter must be set to zero.

#### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer.

If the output buffer is too small, the call fails, [WSAGetLastError](#) returns [WSAEINVAL](#), and the *lpcbBytesReturned* parameter points to a **DWORD** value of zero.

If *lpOverlapped* is **NULL**, the **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned on a successful call cannot be zero.

If the *lpOverlapped* parameter is not **NULL** for overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned may be zero since the size of the data stored can't be determined until the overlapped operation has completed. The final completion status can be retrieved when the appropriate completion method is signaled when the operation has completed.

#### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAIoctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

#### **lpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

#### **lpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

#### **lpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAIoctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAIoctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
<b>WSA_IO_PENDING</b>	Overlapped I/O operation is in progress. This value is returned if an overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSA_OPERATION_ABORTED</b>	The I/O operation has been aborted because of either a thread exit or an application request. This error is returned if an overlapped operation was canceled due to the closure of the socket or the execution of the <b>SIO_FLUSH</b> IOCTL command.
<b>WSAEACCES</b>	An attempt was made to access a socket in a way forbidden by its access permissions. This error is returned under several conditions for persistent port reservations that include the following: the user lacks the required administrative privileges on the local computer or the application is not running in an enhanced shell as the built-in Administrator ( <code>RunAs administrator</code> ).
<b>WSAEFAULT</b>	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if the <i>lpvInBuffer</i> , <i>lpvoutBuffer</i> , <i>lpcbBytesReturned</i> , <i>lpOverlapped</i> or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.
<b>WSAEINPROGRESS</b>	A blocking operation is currently executing. This error is returned if the function is invoked when a callback is in progress.
<b>WSAEINTR</b>	A blocking operation was interrupted by a call to <b>WSACancelBlockingCall</b> . This error is returned if a blocking operation was interrupted.
<b>WSAEINVAL</b>	An invalid argument was supplied. This error is returned if the <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified.
<b>WSAENETDOWN</b>	A socket operation encountered a dead network. This error is returned if the network subsystem has failed.
<b>WSAENOTSOCK</b>	An operation was attempted on something that is not a socket. This error is returned if the descriptor <i>s</i> is not a socket.
<b>WSAEOPNOTSUPP</b>	The attempted operation is not supported for the type of object referenced. This error is returned if the specified IOCTL command is not supported. This error is also returned if the <b>SIO_LOOPBACK_FAST_PATH</b> IOCTL is not supported by the transport provider.

## Remarks

An application can use the **SIO\_LOOPBACK\_FAST\_PATH** IOCTL to reduce the latency and improve the performance of loopback operations on a TCP socket. This IOCTL requests that the TCP/IP stack uses a special fast path for loopback operations on this socket. The **SIO\_LOOPBACK\_FAST\_PATH** IOCTL can be used only with TCP sockets. This IOCTL must be used on both sides of the loopback session. The TCP loopback fast path is

supported using either the IPv4 or IPv6 loopback interface.

The socket that plans to initiate the connection request must apply this IOCTL before making the connection request. So the socket used by the [connect](#), [ConnectEx](#), [WSAConnect](#), [WSAConnectByList](#), or [WSAConnectByName](#) function to initiate the connection must apply this IOCTL to use the fast path for loopback operations.

The socket that is listening for the connection request must apply this IOCTL before accepting the connection. So the socket used by the listen function must apply this IOCTL so any sockets that are accepted will use the fast path for loopback. Any sockets returned by the listen function and passed to the [accept](#), [AcceptEx](#), or [WSAAccept](#) function will be marked to use the special fast path for loopback operations.

Once an application establishes the connection on a loopback interface using the fast path, all packets for the lifetime of the connection must use the fast path.

Applying **SIO\_LOOPBACK\_FAST\_PATH** to a socket which will be connected to a non-loopback path will have no effect.

This TCP loopback optimization results in packets that flow through the Transport Layer (TL) instead of the traditional loopback through the Network Layer. This optimization improves the latency for loopback packets. Once an application opts in for a connection level setting to use the loopback fast path, all packets will follow the loopback path. For loopback communications, congestion and packet drop are not expected. The notion of congestion control and reliable delivery in TCP will be unnecessary. This, however, is not true for flow control. Without flow control, the sender can overwhelm the receive buffer, leading to erroneous TCP loopback behavior. The flow control in the TCP optimized loopback path is maintained by placing send requests in a queue. When receive buffer is full, the TCP/IP stack guarantees that the sends won't complete until the queue is serviced maintaining flow control.

TCP fast path loopback connections in the presence of a Windows Filtering Platform (WFP) callout for connection data must take the un-optimized slow path for loopback. So WFP filters will prevent this new loopback fast path from being used. When a WFP filter is enabled, the system will use the slow path even if the **SIO\_LOOPBACK\_FAST\_PATH** IOCTL was set. This ensures that user-mode applications have the full WFP security capability.

By default, **SIO\_LOOPBACK\_FAST\_PATH** is disabled.

Only a subset of the TCP/IP socket options are supported when the **SIO\_LOOPBACK\_FAST\_PATH** IOCTL is used to enable the loopback fast path on a socket. The list of supported options includes the following:

- **IP\_TTL**
- **IP\_UNICAST\_IF**
- **IPV6\_UNICAST\_HOPS**
- **IPV6\_UNICAST\_IF**
- **IPV6\_V6ONLY**
- **SO\_CONDITIONAL\_ACCEPT**
- **SO\_EXCLUSIVEADDRUSE**
- **SO\_PORT\_SCALABILITY**
- **SO\_RCVBUF**
- **SO\_REUSEADDR**
- **TCP\_BSDURGENT**

## See also

[IPPROTO\\_IP Socket Options](#)

[IPPROTO\\_IPV6 Socket Options](#)

[IPPROTO\\_TCP Socket Options](#)

[socket](#)

[SOL\\_SOCKET Socket Options](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_QUERY\_RSS\_PROCESSOR\_INFO Control Code

3/5/2021 • 4 minutes to read • [Edit Online](#)

## Description

The SIO\_QUERY\_RSS\_PROCESSOR\_INFO control code queries the association between a socket and an RSS processor core and NUMA node.

To perform this operation, call the [WSAIoctl](#) or [WSPIoctl](#) function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_QUERY_RSS_PROCESSOR_INFO, // dwIoControlCode
    NULL,                // lpvInBuffer
    0,                   // cbInBuffer
    (LPVOID) lpvOutBuffer, // output buffer
    (DWORD) cbOutBuffer,  // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_QUERY_RSS_PROCESSOR_INFO, // dwIoControlCode
    NULL,                // lpvInBuffer
    0,                   // cbInBuffer
    (LPVOID) lpvOutBuffer, // output buffer
    (DWORD) cbOutBuffer,  // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);
```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use SIO\_QUERY\_RSS\_PROCESSOR\_INFO for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter is unused for this operation.

### cbInBuffer

The size, in bytes, of the input buffer. This parameter is unused for this operation.

### lpvOutBuffer

A pointer to the output buffer. This parameter should point to a [SOCKET\\_PROCESSOR\\_AFFINITY](#) structure if the *lpOverlapped* and *lpCompletionRoutine* parameters are **NULL**.

### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter must be at least the size of a [SOCKET\\_PROCESSOR\\_AFFINITY](#) structure.

### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer.

If the output buffer is too small, the call fails, [WSAGetLastError](#) returns [WSAEINVAL](#), and the *lpcbBytesReturned* parameter points to a *DWORD* value of zero.

If *lpOverlapped* is **NULL**, the *DWORD* value pointed to by the *lpcbBytesReturned* parameter that is returned on a successful call cannot be zero.

If the *lpOverlapped* parameter is not **NULL** for overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The *DWORD* value pointed to by the *lpcbBytesReturned* parameter that is returned may be zero since the size of the data stored can't be determined until the overlapped operation has completed. The final completion status can be retrieved when the appropriate completion method is signaled when the operation has completed.

### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAIoctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

### **lpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

### **lpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

### **lpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAIoctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAIoctl](#) or [WSPIoctl](#) function returns [SOCKET\\_ERROR](#). To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
ERROR_INSUFFICIENT_BUFFER	The data area passed to a system call is too small. This error is returned if the buffer pointed to by the <i>lpvOutBuffer</i> parameter with a buffer size passed in the <i>cbOutBuffer</i> parameter is too small. The buffer size required will be returned in the <i>lpcbBytesReturned</i> parameter. This error is returned if the <i>cbOutBuffer</i> parameter is less than the size of a <a href="#">SOCKET_PROCESSOR_AFFINITY</a> structure.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	An overlapped operation was canceled due to the closure of the socket or the execution of the SIO_FLUSH IOCTL command.
WSAEFAULT	The <i>lpvInBuffer</i> , <i>lpvoutBuffer</i> , <i>lpcbBytesReturned</i> , <i>lpOverlapped</i> or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAEINTR	A blocking operation was interrupted.
WSAEINVAL	The <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified. This error is returned if the <i>cbOutBuffer</i> parameter is less than the size of a <a href="#">SOCKET_PROCESSOR_AFFINITY</a> structure.
WSAENETDOWN	The network subsystem has failed.
WSAENOPROTOOPT	The socket option is not supported on the specified protocol.
WSAENOTCONN	The socket <i>s</i> is not connected.
WSAENOTSOCK	The descriptor <i>s</i> is not a socket.
WSAEOPNOTSUPP	The specified IOCTL command is not supported. This error is returned if the SIO_QUERY_RSS_PROCESSOR_INFO IOCTL is not supported by the transport provider.

## Remarks

The SIO\_QUERY\_RSS\_PROCESSOR\_INFO IOCTL is supported on Windows 8, and Windows Server 2012, and later versions of the operating system.

The SIO\_QUERY\_RSS\_PROCESSOR\_INFO IOCTL is used to determine the association between a socket and an RSS processor core and NUMA node. This IOCTL returns a [SOCKET\\_PROCESSOR\\_AFFINITY](#) structure that contains the [PROCESSOR\\_NUMBER](#) and the NUMA node ID. The returned [PROCESSOR\\_NUMBER](#) structure contains a group number and relative processor number within the group.

If the socket is a UDP socket, the socket must be connected for the SIO\_QUERY\_RSS\_PROCESSOR\_INFO IOCTL to work properly.

## See also

[PROCESSOR\\_NUMBER](#)

[socket](#)

[SOCKET\\_PROCESSOR\\_AFFINITY](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_QUERY\_TRANSPORT\_SETTING Control Code

3/5/2021 • 5 minutes to read • [Edit Online](#)

## Description

The SIO\_QUERY\_TRANSPORT\_SETTING control code queries the transport settings on a socket.

To perform this operation, call the [WSAIoctl](#) or [WSPIoctl](#) function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_QUERY_TRANSPORT_SETTING, // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to the input buffer
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    (LPVOID) lpvOutBuffer, // pointer to the output buffer
    (DWORD) cbOutBuffer,  // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);

```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_QUERY_TRANSPORT_SETTING, // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to the input buffer
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    (LPVOID) lpvOutBuffer, // pointer to the output buffer
    (DWORD) cbOutBuffer,  // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);

```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use SIO\_QUERY\_TRANSPORT\_SETTING for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter contains a pointer to a structure where the first member of the structure is a [TRANSPORT\\_SETTING\\_ID](#) structure that determines what transport setting is being queried.

### cbInBuffer

The size, in bytes, of the input buffer. This parameter depends on the transport setting being queried.

### lpvOutBuffer

A pointer to the output buffer. This parameter depends on the transport setting being queried if the *lpOverlapped* and *lpCompletionRoutine* parameters are **NULL**.

### **cbOutBuffer**

The size, in bytes, of the output buffer.

### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer.

If the output buffer is too small, the call fails, [WSAGetLastError](#) returns [WSAEINVAL](#), and the *lpcbBytesReturned* parameter points to a **DWORD** value of zero.

If *lpOverlapped* is **NULL**, the **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned on a successful call cannot be zero.

If the *lpOverlapped* parameter is not **NULL** for overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned may be zero since the size of the data stored can't be determined until the overlapped operation has completed. The final completion status can be retrieved when the appropriate completion method is signaled when the operation has completed.

### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAOctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

### **lpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

### **lpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

### **lpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAOctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAOctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
ERROR_INSUFFICIENT_BUFFER	The data area passed to a system call is too small. This error is returned if the buffer pointed to by the <i>lpvOutBuffer</i> parameter with a buffer size passed in the <i>cbOutBuffer</i> parameter is too small. The buffer size required will be returned in the <i>lpcbBytesReturned</i> parameter.
WSA_IO_PENDING	An overlapped operation was successfully initiated and completion will be indicated at a later time.
WSA_OPERATION_ABORTED	An overlapped operation was canceled due to the closure of the socket or the execution of the SIO_FLUSH IOCTL command.
WSAEFAULT	The <i>lpvOutBuffer</i> , <i>lpcbBytesReturned</i> , <i>lpOverlapped</i> , or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAEINTR	A blocking operation was interrupted.
WSAEINVAL	The <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified.
WSAENETDOWN	The network subsystem has failed.
WSAENOPROTOOPT	The socket option is not supported on the specified protocol.
WSAENOTCONN	The socket s is not connected.
WSAENOTSOCK	The descriptor s is not a socket.
WSAEOPNOTSUPP	The specified IOCTL command is not supported. This error is returned if the SIO_QUERY_TRANSPORT_SETTING IOCTL is not supported by the transport provider.

## Remarks

The SIO\_QUERY\_TRANSPORT\_SETTING IOCTL is supported on Windows 8, and Windows Server 2012, and later versions of the operating system.

The SIO\_QUERY\_TRANSPORT\_SETTING IOCTL is a generic IOCTL used to query the transport settings on a socket. The transport setting being queried is based on the [TRANSPORT\\_SETTING\\_ID](#) passed in the *lpvInBuffer* parameter.

The only transport setting currently defines is for the REAL\_TIME\_NOTIFICATION\_CAPABILITY capability on a TCP socket.

If the [TRANSPORT\\_SETTING\\_ID](#) passed in the *lpvInBuffer* parameter has the Guid member set to REAL\_TIME\_NOTIFICATION\_CAPABILITY, then this is a request to query the real time notification settings for the TCP socket used with the [ControlChannelTrigger](#) to receive background network notifications in a Windows Store app. The *lpvInBuffer* parameter should point to a [TRANSPORT\\_SETTING\\_ID](#) structure. The

*lpvOutBuffer* parameter should point to a **REAL\_TIME\_NOTIFICATION\_SETTING\_OUTPUT** structure. This transport setting applies only to TCP sockets. This transport setting provides a way for WinInet or similar network services to query a given TCP socket to determine the **ControlChannelTrigger** status. A Windows Store app will not call this IOCTL directly. If the **WSAIoctl** or **WSPIoctl** call is successful, this IOCTL returns a **REAL\_TIME\_NOTIFICATION\_SETTING\_OUTPUT** structure with the current status.

The **SIO\_QUERY\_TRANSPORT\_SETTING** IOCTL provides a way for WinInet or similar network services to query for the transport setting status for a given TCP socket to determine if **ControlChannelTrigger** is enabled on the socket. A Windows Store app will not call this IOCTL directly.

This IOCTL applies only to TCP sockets.

## See also

[CONTROL\\_CHANNEL\\_TRIGGER\\_STATUS](#)

[ControlChannelTrigger](#)

[REAL\\_TIME\\_NOTIFICATION\\_SETTING\\_OUTPUT](#)

[SIO\\_APPLY\\_TRANSPORT\\_SETTING](#)

[socket](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_RECORDS Control Code

3/5/2021 • 7 minutes to read • [Edit Online](#)

## Description

The **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_RECORDS** control code retrieves the redirect record for the accepted TCP/IP connection for use by a Windows Filtering Platform (WFP) redirect service.

To perform this operation, call the **WSAIoctl** or **WSPIoctl** function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS, // dwIoControlCode
    NULL,                // lpvInBuffer
    0,                   // cbInBuffer
    (LPVOID) lpvOutBuffer, // output buffer
    (DWORD) cbOutBuffer,  // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS, // dwIoControlCode
    NULL,                // lpvInBuffer
    0,                   // cbInBuffer
    (LPVOID) lpvOutBuffer, // output buffer
    (DWORD) cbOutBuffer,  // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_RECORDS** for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter is unused for this operation.

### cbInBuffer

The size, in bytes, of the input buffer. This parameter is unused for this operation.

### lpvOutBuffer

A pointer to the output buffer. This parameter should point to a **ULONG** data type if the *lpOverlapped* and *lpCompletionRoutine* parameters are **NULL**.

### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter must be at least the size of a **ULONG** data type.

### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer.

If the output buffer is too small, the call fails, [WSAGetLastError](#) returns [WSAEINVAL](#), and the *lpcbBytesReturned* parameter points to a **DWORD** value of zero.

If *lpOverlapped* is **NULL**, the **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned on a successful call cannot be zero.

If the *lpOverlapped* parameter is not **NULL** for overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned may be zero since the size of the data stored can't be determined until the overlapped operation has completed. The final completion status can be retrieved when the appropriate completion method is signaled when the operation has completed.

### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAOctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

### **lpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

### **lpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

### **lpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAOctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAOctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
<b>ERROR_INSUFFICIENT_BUFFER</b>	The data area passed to a system call is too small. This error is returned if the buffer pointed to by the <i>lpvOutBuffer</i> parameter with a buffer size passed in the <i>cbOutBuffer</i> parameter is too small. The buffer size required will be returned in the <i>lpcbBytesReturned</i> parameter.
<b>WSA_IO_PENDING</b>	An overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSA_OPERATION_ABORTED</b>	An overlapped operation was canceled due to the closure of the socket or the execution of the <b>SIO_FLUSH</b> IOCTL command.
<b>WSAEFAULT</b>	The <i>lpvOutBuffer</i> , <i>lpcbBytesReturned</i> , <i>lpOverlapped</i> , or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.
<b>WSAEINPROGRESS</b>	The function is invoked when a callback is in progress.
<b>WSAEINTR</b>	A blocking operation was interrupted.
<b>WSAEINVAL</b>	The <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified. This error is returned if the <i>cbOutBuffer</i> parameter is less than the size of a <b>ULONG</b> data type.
<b>WSAENETDOWN</b>	The network subsystem has failed.
<b>WSAENOPROTOOPT</b>	The socket option is not supported on the specified protocol.
<b>WSAENOTCONN</b>	The socket <i>s</i> is not connected.
<b>WSAENOTSOCK</b>	The descriptor <i>s</i> is not a socket.
<b>WSAEOPNOTSUPP</b>	The specified IOCTL command is not supported. This error is returned if the <b>SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS</b> IOCTL is not supported by the transport provider.

## Remarks

The **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_RECORDS** IOCTL is supported on Windows 8, and Windows Server 2012, and later versions of the operating system.

WFP allows access to the TCP/IP packet processing path, wherein outgoing and incoming packets can be examined or changed before allowing them to be processed further. By tapping into the TCP/IP processing path, independent software vendors (ISVs) can more easily create firewalls, antivirus software, diagnostic software, and other types of applications and services. WFP provides user-mode and kernel-mode components so that third-party ISVs can participate in the filtering decisions that take place at several layers in the TCP/IP protocol stack and throughout the operating system. The WFP connection redirect feature allows a WFP callout kernel driver to redirect a connection locally to a user-mode process, perform content inspection in user-mode, and forward the inspected content using a different connection to the original destination.

The `SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS` IOCTL and several other related IOCTLS are user-mode components used to allow multiple WFP-based connection proxy applications to inspect the same traffic flow in a cooperative manner. Each inspection agent can safely re-inspect network traffic that has already been inspected by another inspection agent. With the presence of multiple proxies (developed by different ISVs, for example) the connection used by one proxy to communicate with the final destination could in turn be redirected by a second proxy, and that new connection could again be redirected by the original proxy. Without connection tracking, the original connection might never reach its final destination as it gets stuck in an infinite proxy loop.

The `SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS` IOCTL is used to provide proxied connection tracking on redirected socket connections. This WFP feature facilitates tracking of redirection records from the initial redirect of a connection to the final connection to the destination.

The `SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS` IOCTL is used by a WFP-based redirect service to retrieve the redirect record from the accepted TCP/IP packet connection (the connected socket for a TCP socket or a UDP socket, for example) redirected to it by its companion kernel-mode callout registered at `ALE_CONNECT_REDIRECT` layers in a kernel-mode driver. The `SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT` IOCTL is used by a WFP-based redirect service to retrieve the redirect context for a redirect record from the accepted TCP/IP packet connection (the connected socket for a TCP socket or a UDP socket, for example) redirected to it by its companion callout registered at `ALE_CONNECT_REDIRECT` layers. The redirect context is an optional driver-allocated context used if the current redirection state of a connection is that the connection was redirected by the calling redirect service or the connection was previously redirected by the calling redirect service but later redirected again by a different redirect service. For a TCP proxy connection, the redirect service transfers the retrieved redirect record to the TCP socket it uses to proxy the original content using the `SIO_SET_WFP_CONNECTION_REDIRECT_RECORDS` IOCTL.

When the redirect service is redirecting a non-TCP socket (UDP, for example), the redirect records returned by the `SIO_QUERY_WFP_CONNECTION_REDIRECT_RECORDS` IOCTL indicate this to the redirect service using the `WSAMSG` structure used with the `LPFN_WSARECVMSG (WSARecvMsg)` function. The Control member of the `WSAMSG` structure would have a `cmsg_type` in the `WSACMSGHDR` structure set to `IP_CONNECTION_REDIRECT_RECORD`. The `LPFN_WSARECVMSG (WSARecvMsg)` parameter must be supplied by the redirect service when accepting non-TCP redirects.

For non-TCP traffic, the redirect record is forwarded using the `WSAMSG` structure with the `WSASendMsg` function.

Note that for non-TCP traffic, only the flow-creating packet will carry the `IP_CONNECTION_REDIRECT_RECORD`. As a result, only the first proxied packet needs to include this information using the `LPFN_WSARECVMSG (WSARecvMsg)` function.

Since the WFP redirect record is a variable length data blob, the caller can either supply a large output buffer (a 1,024 byte buffer pointed to by the `lpvOutBuffer` parameter, for example) or can pass an output buffer size in the `cbOutBuffer` parameter of 0 to query the buffer size required to hold the returned blob. If the output buffer size is not sufficient to hold the data, the `WSAOctl` or `WSPIoctl` functions will return `ERROR_INSUFFICIENT_BUFFER` and the required buffer size will be returned in value pointed to by the `lpcbBytesReturned` parameter.

The application calling the `SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT` does not need to understand the blob containing the redirect context retrieved. This is an opaque blob of data that the application needs to retrieve and pass back to the new socket.

## See also

[IPPROTO\\_IP Socket Options](#)

[SIO\\_QUERY\\_WFP\\_CONNECTION\\_REDIRECT\\_CONTEXT](#)

[socket](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAMSG](#)

[WSAOVERLAPPED](#)

[LPFN\\_WSARECVMSG \(WSARecvMsg\)](#)

[WSASendMsg](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_CONTEXT

## Control Code

3/5/2021 • 7 minutes to read • [Edit Online](#)

## Description

The **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_CONTEXT** control code retrieves the redirect context for a redirect record used by a Windows Filtering Platform (WFP) redirect service.

To perform this operation, call the **WSAOctl** or **WSPIoctl** function with the following parameters.

```
(socket) s,           // descriptor identifying a socket
SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT, // dwIoControlCode
NULL,                // lpvInBuffer
0,                  // cbInBuffer
(LPVOID) lpvOutBuffer, // output buffer
(DWORD) cbOutBuffer, // size of output buffer
(LPDWORD) lpcbBytesReturned, // number of bytes returned
(LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
(LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
;
```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT, // dwIoControlCode
    NULL,                // lpvInBuffer
    0,                  // cbInBuffer
    (LPVOID) lpvOutBuffer, // output buffer
    (DWORD) cbOutBuffer, // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);
```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_CONTEXT** for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter is unused for this operation.

### cbInBuffer

The size, in bytes, of the input buffer. This parameter is unused for this operation.

### lpvOutBuffer

A pointer to the output buffer. This parameter should point to a **ULONG** data type if the *lpOverlapped* and

*lpCompletionRoutine* parameters are **NULL**.

#### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter must be at least the size of a **ULONG** data type.

#### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer.

If the output buffer is too small, the call fails, [WSAGetLastError](#) returns [WSAEINVAL](#), and the *lpcbBytesReturned* parameter points to a **DWORD** value of zero.

If *lpOverlapped* is **NULL**, the **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned on a successful call cannot be zero.

If the *lpOverlapped* parameter is not **NULL** for overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned may be zero since the size of the data stored can't be determined until the overlapped operation has completed. The final completion status can be retrieved when the appropriate completion method is signaled when the operation has completed.

#### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAOctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

#### **lpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

#### **lpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

#### **lpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAOctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAOctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
<b>WSA_IO_PENDING</b>	An overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSA_OPERATION_ABORTED</b>	An overlapped operation was canceled due to the closure of the socket or the execution of the SIO_FLUSH IOCTL command.
<b>WSAEFAULT</b>	The <i>lpvOutBuffer</i> , <i>lpcbBytesReturned</i> , <i>lpOverlapped</i> , or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.
<b>WSAEINPROGRESS</b>	The function is invoked when a callback is in progress.
<b>WSAEINTR</b>	A blocking operation was interrupted.
<b>WSAEINVAL</b>	The <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified. This error is returned if the <i>cbOutBuffer</i> parameter is less than the size of a <b>ULONG</b> data type.
<b>WSAENETDOWN</b>	The network subsystem has failed.
<b>WSAENOPROTOOPT</b>	The socket option is not supported on the specified protocol.
<b>WSAENOTCONN</b>	The socket <i>s</i> is not connected.
<b>WSAENOTSOCK</b>	The descriptor <i>s</i> is not a socket.
<b>WSAEOPNOTSUPP</b>	The specified IOCTL command is not supported. This error is returned if the <b>SIO_QUERY_WFP_CONNECTION_REDIRECT_CONTEXT</b> IOCTL is not supported by the transport provider.

## Remarks

The **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_CONTEXT** IOCTL is supported on Windows 8, and Windows Server 2012, and later versions of the operating system.

WFP allows access to the TCP/IP packet processing path, wherein outgoing and incoming packets can be examined or changed before allowing them to be processed further. By tapping into the TCP/IP processing path, independent software vendors (ISVs) can more easily create firewalls, antivirus software, diagnostic software, and other types of applications and services. WFP provides user-mode and kernel-mode components so that third-party ISVs can participate in the filtering decisions that take place at several layers in the TCP/IP protocol stack and throughout the operating system. The WFP connection redirect feature allows a WFP callout kernel driver to redirect a connection locally to a user-mode process, perform content inspection in user-mode, and forward the inspected content using a different connection to the original destination.

The **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_CONTEXT** IOCTL and several other related IOCTLS are user-mode components used to allow multiple WFP-based connection proxy applications to inspect the same traffic flow in a cooperative manner. Each inspection agent can safely re-inspect network traffic that has already been inspected by another inspection agent. With the presence of multiple proxies (developed by different ISVs, for example) the connection used by one proxy to communicate with the final destination could in turn be

redirected by a second proxy, and that new connection could again be redirected by the original proxy. Without connection tracking, the original connection might never reach its final destination as it gets stuck in an infinite proxy loop.

The **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_CONTEXT** IOCTL is used to allow multiple WFP-based connection proxy applications to inspect the same traffic flow in a cooperative manner. Each inspection agent can safely re-inspect network traffic that has already been inspected by another inspection agent. With the presence of multiple proxies (developed by different ISVs, for example) the connection used by one proxy to communicate with the final destination could in turn be redirected by a second proxy, and that new connection could again be redirected by the original proxy. Without connection tracking, the original connection might never reach its final destination as it gets stuck in an infinite proxy loop. The **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_CONTEXT** IOCTL is used to provide proxied connection tracking on redirected socket connections. This WFP feature facilitates tracking of redirection records from the initial redirect of a connection to the final connection to the destination.

The **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_RECORDS** IOCTL is used by a WFP-based redirect service to retrieve the redirect record from the accepted TCP/IP packet connection (the connected socket for a TCP socket or a UDP socket, for example) redirected to it by its companion kernel-mode callout registered at **ALE\_CONNECT\_REDIRECT** layers in a kernel-mode driver. The **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_CONTEXT** IOCTL is used by a WFP-based redirect service to retrieve the redirect context for a redirect record from the accepted TCP/IP packet connection (the connected socket for a TCP socket or a UDP socket, for example) redirected to it by its companion callout registered at **ALE\_CONNECT\_REDIRECT** layers. The redirect context is optional and is used if the current redirection state of a connection is that the connection was redirected by the calling redirect service or the connection was previously redirected by the calling redirect service but later redirected again by a different redirect service. The redirect service transfers the retrieved redirect record to the TCP socket it uses to proxy the original content using the **SIO\_SET\_WFP\_CONNECTION\_REDIRECT\_RECORDS** IOCTL.

Since the WFP redirect context is a variable length data blob set by the redirect service, the caller can either supply a large output buffer (a 1K buffer pointed to by the *lpvOutBuffer* parameter, for example) or can pass as an output buffer size in the *cbOutBuffer* parameter of 0 to query the buffer size required to hold the returned blob. If the output buffer size is not sufficient to hold the data, the **WSAOctl** or **WSPIoctl** functions will return **ERROR\_INSUFFICIENT\_BUFFER** and the required buffer size will be returned in value pointed to by the *lpcbBytesReturned* parameter.

The application calling the **SIO\_QUERY\_WFP\_CONNECTION\_REDIRECT\_RECORDS** IOCTL does not need to understand the blob containing the redirect records retrieved. This is an opaque blob of data that the application needs to retrieve and pass back to the new socket.

## See also

[IPPROTO\\_IP Socket Options](#)

[SIO\\_QUERY\\_WFP\\_CONNECTION\\_REDIRECT\\_RECORDS](#)

[SIO\\_SET\\_WFP\\_CONNECTION\\_REDIRECT\\_RECORDS](#)

[socket](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAOctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

**WSASocketW**

# SIO\_RCVALL Control Code

3/5/2021 • 7 minutes to read • [Edit Online](#)

## Description

The SIO\_RCVALL control code enables a socket to receive all IPv4 or IPv6 packets passing through a network interface.

To perform this operation, call the [WSAIoctl](#) or [WSPIoctl](#) function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_RCV_ALL,         // dwIoControlCode
    NULL,                // lpvInBuffer0,
    NULL,                // lpvOutBuffer output buffer
    (DWORD) cbOutBuffer, // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_RCV_ALL,         // dwIoControlCode
    NULL,                // lpvInBuffer
    0,                  // cbInBuffer
    NULL,                // lpvOutBuffer output buffer
    (DWORD) cbOutBuffer, // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use **SIO\_RCVALL** for this operation.

### lpvInBuffer

A pointer to the input buffer that should contain the option value. The possible values for the SIO\_RCVALL IOCTL option are specified in the **RCVALL\_VALUE** enumeration defined in the *Mstcpip.h* header file.

The possible values for **SIO\_RCVALL** are as follows:

VALUE	MEANING
RCVALL_OFF	Disable this option so a socket does not receive all IPv4 or IPv6 packets passing through a network interface.

VALUE	MEANING
RCVALL_ON	Enable this option so a socket receives all IPv4 or IPv6 packets passing through a network interface. This option enables promiscuous mode on the network interface card (NIC), if the NIC supports promiscuous mode. On a LAN segment with a network hub, a NIC that supports promiscuous mode will capture all IPv4 or IPv6 traffic on the LAN, including traffic between other computers on the same LAN segment. All of the captured packets (IPv4 or IPv6, depending on the socket) will be delivered to the raw socket. This option will not capture other packets (ARP, IPX, and NetBEUI packets, for example) on the interface. Netmon uses the same mode for the network interface, but does not use this option to capture traffic.
RCVALL_SOCKETLEVELONLY	This feature is not currently implemented, so setting this option does not have any affect.
RCVALL_IPLEVEL	Enable this option so an IPv4 or IPv6 socket receives all packets at the IP level passing through a network interface. This option does not enable promiscuous mode on the network interface card. This option only affects packet processing at the IP level. The NIC still receives only packets directed to its configured unicast and multicast addresses. However, a socket with this option enabled will receive not only packets directed to specific IP addresses, but will receive all the IPv4 or IPv6 packets the NIC receives. This option will not capture other packets (ARP, IPX, and NetBEUI packets, for example) received on the interface.

#### **cbInBuffer**

The size, in bytes, of the input buffer. This parameter should be equal to or greater than the size of the RCVALL\_VALUE enumeration value pointed to by the *lpvInBuffer* parameter.

#### **IpvOutBuffer**

A pointer to the output buffer. This parameter is unused for this operation.

#### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter is unused for this operation.

#### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer. This parameter is unused for this operation.

#### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAIoctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

#### **IpCompletionRoutine**

Type: \_In\_opt\_ **LWWSAOVERLAPPED\_COMPLETION\_ROUTINE**

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

#### **lpThreadId**

A pointer to a **WSATHREADID** structure to be used by the provider in a subsequent call to **WPUQueueApc**. The provider should store the referenced **WSATHREADID** structure (not the pointer to same) until after the **WPUQueueApc** function returns.

**Note** This parameter applies only to the **WSPIoctl** function.

#### **lpErrno**

A pointer to the error code.

**Note** This parameter applies only to the **WSPIoctl** function.

## Return value

If the operation completes successfully, the **WSAIoctl** or **WSPIoctl** function returns zero.

If the operation fails or is pending, the **WSAIoctl** or **WSPIoctl** function returns **SOCKET\_ERROR**. To get extended error information, call **WSAGetLastError**.

ERROR CODE	MEANING
<b>WSA_IO_PENDING</b>	An overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSA_OPERATION_ABORTED</b>	An overlapped operation was canceled due to the closure of the socket or the execution of the <b>SIO_FLUSH</b> IOCTL command.
<b>WSAEFAULT</b>	The <i>lpOverlapped</i> or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.
<b>WSAEINPROGRESS</b>	The function is invoked when a callback is in progress.
<b>WSAEINTR</b>	A blocking operation was interrupted.
<b>WSAEINVAL</b>	The <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified. This error is also returned if the <i>cblnBuffer</i> parameter is less than the <code>sizeof(UCHAR)</code> or the <i>lpvlnBuffer</i> parameter points to value that is not a <b>RCVALL_VALUE</b> enumeration value. This error can also be returned if the network interface associated with the socket cannot be found. This could occur if the network interface associated with the socket is deleted or removed (a remove PCMCIA or USB network device, for example).
<b>WSAENETDOWN</b>	The network subsystem has failed.
<b>WSAENOBUFS</b>	No buffer space available.

ERROR CODE	MEANING
WSAENOPROTOOPT	The socket option is not supported on the specified protocol.
WSAENOTSOCK	The descriptor <i>s</i> is not a socket.
WSAEOPNOTSUPP	The specified IOCTL command is not supported. This error is returned if the SIO_RCVALL IOCTL is not supported by the transport provider.

## Remarks

The SIO\_RCVALL IOCTL is supported on Windows 2000 and later versions of the operating system.

The SIO\_RCVALL IOCTL enables a socket to receive all IPv4 or IPv6 packets on a network interface. The socket handle passed to the [WSAOctl](#) or [WSPIoctl](#) function must be one of the following:

- An IPv4 socket that was created with the address family set to AF\_INET, the socket type set to SOCK\_RAW, and the protocol set to IPPROTO\_IP.
- An IPv6 socket that was created with the address family set to AF\_INET6, the socket type set to SOCK\_RAW, and the protocol set to IPPROTO\_IPV6.

For more information on raw sockets, see [TCP/IP Raw Sockets](#).

The socket also must be bound to an explicit local IPv4 or IPv6 interface, which means that you cannot bind to INADDR\_ANY or in6addr\_any.

Once the socket is bound and the IOCTL completes successfully, calls to the [WSARecv](#) or [recv](#) functions return IPv4 datagrams passing through the given IPv4 interface or return IPv6 datagrams passing through the given IPv6 interface. Note that you must supply a sufficiently large buffer. Setting this IOCTL will only capture IPv4 or IPv6 packets on a given interface. This IOCTL will not capture other packets (ARP, IPX, and NetBEUI packets, for example) on the interface.

A socket bound to a specific local interface with the SIO\_RCVALL IOCTL will receive only packets passing through that interface. It will not receive packets received on another interface and getting forwarded out on another interface different from the socket bound with SIO\_RCVALL IOCTL.

On Windows Server 2008 and earlier, the SIO\_RCVALL IOCTL setting would not capture local packets sent out of a network interface. This included packets received on another interface and forwarded out the network interface specified for the SIO\_RCVALL IOCTL.

On Windows 7 and Windows Server 2008 R2 , this was changed so that local packets sent out of a network interface are also captured. This includes packets received on another interface and then forwarded out the network interface bound to the socket with SIO\_RCVALL IOCTL.

Setting this IOCTL requires Administrator privilege on the local computer.

This feature is sometimes referred to as promiscuous mode. Any direct change from applying this option on one interface and then to another interface with a single call using this IOCTL is not supported. An application must first use this IOCTL to turn off the behavior on the first interface, and then use this IOCTL to enable the behavior on a new interface.

## See also

[socket](#)

[TCP/IP Raw Sockets](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_RELEASE\_PORT\_RESERVATION Control Code

3/5/2021 • 8 minutes to read • [Edit Online](#)

## Description

The **SIO\_RELEASE\_PORT\_RESERVATION** control code releases a runtime reservation for a block of TCP or UDP ports. The runtime reservation to be released must have been obtained from the issuing process using the [SIO\\_ACQUIRE\\_PORT\\_RESERVATION](#) IOCTL.

To perform this operation, call the [WSAIoctl](#) or [WSPIoctl](#) function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_RELEASE_PORT_RESERVATION, // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to a INET_PORT_RESERVATION_TOKEN structure
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    NULL,                // lpvOutBuffer is a pointer to the output buffer
    0,                   // cbOutBuffer is the size, in bytes, of the output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_RELEASE_PORT_RESERVATION, // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to a INET_PORT_RESERVATION_TOKEN structure
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    NULL,                // lpvOutBuffer is a pointer to the output buffer
    0,                   // cbOutBuffer is the size, in bytes, of the output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);
```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use **SIO\_RELEASE\_PORT\_RESERVATION** for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter contains a pointer to an [INET\\_PORT\\_RESERVATION\\_TOKEN](#) structure with the token for the TCP or UDP port reservation to release.

### cbInBuffer

The size, in bytes, of the input buffer. This parameter must be at least the size of the [INET\\_PORT\\_RESERVATION\\_TOKEN](#) structure.

### **IpvOutBuffer**

A pointer to the output buffer. This parameter is unused for this operation.

### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter must be set to zero.

### **IpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer.

If the output buffer is too small, the call fails, [WSAGetLastError](#) returns [WSAEINVAL](#), and the *IpcbBytesReturned* parameter points to a **DWORD** value of zero.

If *IpOverlapped* is **NULL**, the **DWORD** value pointed to by the *IpcbBytesReturned* parameter that is returned on a successful call cannot be zero.

If the *IpOverlapped* parameter is not **NULL** for overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The **DWORD** value pointed to by the *IpcbBytesReturned* parameter that is returned may be zero since the size of the data stored can't be determined until the overlapped operation has completed. The final completion status can be retrieved when the appropriate completion method is signaled when the operation has completed.

### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *IpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *IpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *IpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAOctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

### **IpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

### **IpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

### **IpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAOctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAOctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
<b>WSA_IO_PENDING</b>	Overlapped I/O operation is in progress. This value is returned if an overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSA_OPERATION_ABORTED</b>	The I/O operation has been aborted because of either a thread exit or an application request. This error is returned if an overlapped operation was canceled due to the closure of the socket or the execution of the <b>SIO_FLUSH</b> IOCTL command.
<b>WSAEFAULT</b>	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if the <i>lpOverlapped</i> or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.
<b>WSAEINPROGRESS</b>	A blocking operation is currently executing. This error is returned if the function is invoked when a callback is in progress.
<b>WSAEINTR</b>	A blocking operation was interrupted by a call to <b>WSACancelBlockingCall</b> . This error is returned if a blocking operation was interrupted.
<b>WSAEINVAL</b>	An invalid argument was supplied. This error is returned if the <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified.
<b>WSAENETDOWN</b>	A socket operation encountered a dead network. This error is returned if the network subsystem has failed.
<b>WSAENOTSOCK</b>	An operation was attempted on something that is not a socket. This error is returned if the descriptor <i>s</i> is not a socket.
<b>WSAEOPNOTSUPP</b>	The attempted operation is not supported for the type of object referenced. This error is returned if the specified IOCTL command is not supported. This error is also returned if the <b>SIO_RELEASE_PORT_RESERVATION</b> IOCTL is not supported by the transport provider. This error is also returned when an attempt to use the <b>SIO_RELEASE_PORT_RESERVATION</b> IOCTL is made on a socket other than UDP or TCP.

## Remarks

The **SIO\_RELEASE\_PORT\_RESERVATION** IOCTL is supported on Windows Vista and later versions of the operating system.

Applications and services which need to reserve ports fall into two categories. The first category includes components which need a particular port as part of their operation. Such components will generally prefer to specify their required port at installation time (in an application manifest, for example). The second category includes components which need any available port or block of ports at runtime. These two categories correspond to specific and wildcard port reservation requests. Specific reservation requests may be persistent or runtime, while wildcard port reservation requests are only supported at runtime.

The [SIO\\_ACQUIRE\\_PORT\\_RESERVATION](#) IOCTL is used to request a runtime reservation for a block of TCP or UDP ports. For runtime port reservations, the port pool requires that reservations be consumed from the process on whose socket the reservation was granted. Runtime port reservations last only as long as the lifetime of the socket on which the [SIO\\_ACQUIRE\\_PORT\\_RESERVATION](#) IOCTL was called. In contrast, persistent port reservations created using the [CreatePersistentTcpPortReservation](#) or [CreatePersistentUdpPortReservation](#) function may be consumed by any process with the ability to obtain persistent reservations.

The [SIO\\_RELEASE\\_PORT\\_RESERVATION](#) IOCTL is used to release a runtime reservation for a block of TCP or UDP ports.

If both *lpOverlapped* and *lpCompletionRoutine* parameters 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, except that the function can block if socket *s* is in blocking mode. If socket *s* is in non-blocking mode, this function will still block since this particular IOCTL does not support non-blocking mode.

For overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time.

Any IOCTL may block indefinitely, depending on the service provider's implementation. If the application cannot tolerate blocking in a [WSAIoctl](#) or [WSPIoctl](#) function call, overlapped I/O would be advised for IOCTLs that are especially likely to block.

The [SIO\\_RELEASE\\_PORT\\_RESERVATION](#) IOCTL can fail with [WSAEINTR](#) or [WSA\\_OPERATION\\_ABORTED](#) under the following cases:

- The request is canceled by the I/O Manager.
- The socket is closed.

## Examples

The following example acquires a runtime port reservation and then releases the runtime port reservation.

```
#ifndef UNICODE
#define UNICODE
#endif

#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN
#endif

#include <Windows.h>
#include <winsock2.h>
#include <mstcpip.h>
#include <ws2ipdef.h>
#include <stdio.h>
#include <stdlib.h>

// Need to link with Ws2_32.lib for Winsock functions
#pragma comment(lib, "ws2_32.lib")

int wmain(int argc, WCHAR ** argv)
{

    // Declare and initialize variables

    int startPort = 0;           // host byte order
    int numPorts = 0;
    USHORT startPortns = 0;      // Network byte order

    INET PORT RANGE portRange = { 0 };
```

```

INET_PORT_RESERVATION_INSTANCE portRes = { 0 };

unsigned long status = 0;

WSADATA wsaData = { 0 };
int iResult = 0;

SOCKET sock = INVALID_SOCKET;
int iFamily = AF_INET;
int iType = 0;
int iProtocol = 0;

SOCKET sockRes = INVALID_SOCKET;

DWORD bytesReturned = 0;

// Validate the parameters
if (argc != 6) {
    wprintf
        (L"usage: %s <addressfamily> <type> <protocol> <StartingPort> <NumberOfPorts>\n",
        argv[0]);
    wprintf(L"Opens a socket for the specified family, type, & protocol\n");
    wprintf
        (L"and then acquires a runtime port reservation for the protocol specified\n");
    wprintf(L"%ws example usage\n", argv[0]);
    wprintf(L"    %ws 2 2 17 5000 20\n", argv[0]);
    wprintf(L"    where AF_INET=2 SOCK_DGRAM=2 IPPROTO_UDP=17 StartPort=5000 NumPorts=20", argv[0]);

    return 1;
}

iFamily = _wtoi(argv[1]);
iType = _wtoi(argv[2]);
iProtocol = _wtoi(argv[3]);

startPort = _wtoi(argv[4]);
if (startPort < 0 || startPort > 65535) {
    wprintf(L"Starting point must be either 0 or between 1 and 65,535\n");
    return 1;
}
startPortns = htons((USHORT) startPort);

numPorts = _wtoi(argv[5]);
if (numPorts < 0) {
    wprintf(L"Number of ports must be a positive number\n");
    return 1;
}

portRange.StartPort = startPortns;
portRange.NumberOfPorts = (USHORT) numPorts;

// Initialize Winsock
iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
if (iResult != 0) {
    wprintf(L"WSAStartup failed with error = %d\n", iResult);
    return 1;
}

sock = socket(iFamily, iType, iProtocol);
if (sock == INVALID_SOCKET) {
    wprintf(L"socket function failed with error = %d\n", WSAGetLastError());
    WSACleanup();
    return 1;
} else {
    wprintf(L"socket function succeeded\n");

    iResult =
        WSAIoctl(sock, SIO_ACQUIRE_PORT_RESERVATION, (LPVOID) & portRange,
            sizeof(TNET_PORT_RANGE), (LPVOID) & portRes

```

```

        sizeof( INET_PORT_RANGE), (LPVOID) & portRes,
        sizeof( INET_PORT_RESERVATION_INSTANCE), &bytesReturned, NULL, NULL);
    if (iResult != 0) {
        wprintf(L"WSAIoctl(SIO_ACQUIRE_PORT_RESERVATION) failed with error = %d\n",
               WSAGetLastError());
        closesocket(sock);
        WSACleanup();
        return 1;
    } else {
        wprintf(
            L"WSAIoctl(SIO_ACQUIRE_PORT_RESERVATION) succeeded, bytesReturned = %u\n",
            bytesReturned);
        wprintf(L" Starting port=%d, Number of Ports=%d, Token=%I64d\n",
               htons(portRes.Reservation.StartPort),
               portRes.Reservation.NumberOfPorts, portRes.Token);

        iResult =
            WSAIoctl(sock, SIO_RELEASE_PORT_RESERVATION, (LPVOID) & portRes.Token,
                      sizeof( ULONG64), NULL, 0, &bytesReturned, NULL, NULL);
        if (iResult != 0) {
            wprintf(
                L"WSAIoctl(SIO_RELEASE_PORT_RESERVATION) failed with error = %d\n",
                WSAGetLastError());
        } else {
            wprintf(
                L"WSAIoctl(SIO_RELEASE_PORT_RESERVATION) succeeded, bytesReturned = %u\n",
                bytesReturned);
        }
    }

    if (sock != INVALID_SOCKET) {
        iResult = closesocket(sock);
        if (iResult == SOCKET_ERROR) {
            wprintf(L"closesocket for first socket failed with error = %d\n",
                   WSAGetLastError());
        }
    }
}

WSACleanup();

return 0;
}

```

## See also

[CreatePersistentTcpPortReservation](#)

[CreatePersistentUdpPortReservation](#)

[DeletePersistentTcpPortReservation](#)

[DeletePersistentUdpPortReservation](#)

[INET\\_PORT\\_RESERVATION\\_TOKEN](#)

[LookupPersistentTcpPortReservation](#)

[LookupPersistentUdpPortReservation](#)

[SIO\\_ACQUIRE\\_PORT\\_RESERVATION](#)

[SIO\\_ASSOCIATE\\_PORT\\_RESERVATION](#)

[socket](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_SET\_COMPATIBILITY\_MODE Control Code

3/5/2021 • 15 minutes to read • [Edit Online](#)

## Description

The SIO\_SET\_COMPATIBILITY\_MODE control code requests how the networking stack should handle certain behaviors for which the default way of handling the behavior may differ across Windows versions.

To perform this operation, call the [WSAIoctl](#) or [WSPIoctl](#) function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_SET_COMPATIBILITY_MODE, // dwIoControlCode
    (LPVOID) lpvInBuffer,   // pointer to WSA_COMPATIBILITY_MODE struct
    (DWORD) cbInBuffer,     // length of input buffer
    NULL,                // output buffer
    0,                   // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_SET_COMPATIBILITY_MODE, // dwIoControlCode
    (LPVOID) lpvInBuffer,   // pointer to WSA_COMPATIBILITY_MODE struct
    (DWORD) cbInBuffer,     // length of input buffer
    NULL,                // output buffer
    0,                   // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);
```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use SIO\_SET\_COMPATIBILITY\_MODE for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter should point to a WSA\_COMPATIBILITY\_MODE structure.

### cbInBuffer

The size, in bytes, of the input buffer. This parameter should equal to or greater than the size of the WSA\_COMPATIBILITY\_MODE structure pointed to by the *lpvInBuffer* parameter.

### lpvOutBuffer

A pointer to the output buffer. This parameter is unused for this operation.

### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter must be set to zero.

### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer. This returned parameter points to a **DWORD** value of zero for this operation, since there is no output.

### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAIoctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

### **lpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

### **lpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

### **lpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAIoctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAIoctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
<b>WSA_IO_PENDING</b>	An overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSA_OPERATION_ABORTED</b>	An overlapped operation was canceled due to the closure of the socket or the execution of the <b>SIO_FLUSH</b> IOCTL command.
<b>WSAEFAULT</b>	The <i>lpOverlapped</i> or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.

ERROR CODE	MEANING
WSAEINPROGRESS	The function is invoked when a callback is in progress.
WSAEINTR	A blocking operation was interrupted.
WSAEINVAL	The <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified. This error is returned if the <i>cbInBuffer</i> parameter is less than the sizeof the <b>WSA_COMPATIBILITY_MODE</b> structure.
WSAENETDOWN	The network subsystem has failed.
WSAENOPROTOOPT	The socket option is not supported on the specified protocol.
WSAENOTCONN	The socket <i>s</i> is not connected.
WSAENOTSOCK	The descriptor <i>s</i> is not a socket.
WSAEOPNOTSUPP	The specified IOCTL command is not supported. This error is returned if the <b>SIO_SET_COMPATIBILITY_MODE</b> IOCTL is not supported by the transport provider. This error is also returned when an attempt to use the <b>SIO_SET_COMPATIBILITY_MODE</b> IOCTL is made on a datagram socket.

## Remarks

the **SIO\_SET\_COMPATIBILITY\_MODE** IOCTL requests how the networking stack should handle certain behaviors for which the default way of handling the behavior may differ across Windows versions. The input argument structure for **SIO\_SET\_COMPATIBILITY\_MODE** is specified in the **WSA\_COMPATIBILITY\_MODE** structure defined in the *Mswsockdef.h* header file. A pointer to the **WSA\_COMPATIBILITY\_MODE** structure is passed in the *cbInBuffer* parameter. This structure is defined as follows:

```
// Need to #include <mswsock.h>

/* Argument structure for SIO_SET_COMPATIBILITY_MODE */
typedef struct _WSA_COMPATIBILITY_MODE {
    WSA_COMPATIBILITY_BEHAVIOR_ID BehaviorId;
    ULONG TargetOsVersion;
} WSA_COMPATIBILITY_MODE, *PWSA_COMPATIBILITY_MODE;
```

The value specified in the **BehaviorId** member indicates the behavior requested. The value specified in the **TargetOsVersion** member indicates the Windows version that is being requested for the behavior.

The **BehaviorId** member can be one of the values from the **WSA\_COMPATIBILITY\_BEHAVIOR\_ID** enumeration type defined in the *Mswsockdef.h* header file. The possible values for the **BehaviorId** member are as follows:

TERM	DESCRIPTION

TERM	DESCRIPTION
WsaBehaviorAll	This is equivalent to requesting all of the possible compatible behaviors defined for <b>WSA_COMPATIBILITY_BEHAVIOR_ID</b> .
WsaBehaviorReceiveBuffering	When the <b>TargetOsVersion</b> member is set to a value for Windows Vista or later, reductions to the TCP receive buffer size on this socket using the <b>SO_RCVBUF</b> socket option are allowed even after a TCP connection has been established. When the <b>TargetOsVersion</b> member is set to a value earlier than Windows Vista, reductions to the TCP receive buffer size on this socket using the <b>SO_RCVBUF</b> socket option are not allowed after connection establishment.
WsaBehaviorAutoTuning	When the <b>TargetOsVersion</b> member is set to a value for Windows Vista or later, receive window auto-tuning is enabled and the TCP window scale factor is reduced to 2 from the default value of 8. When the <b>TargetOsVersion</b> is set to a value earlier than Windows Vista, receive window auto-tuning is disabled. The TCP window scaling option is also disabled and the maximum true receive window size is limited to 65,535 bytes. The TCP window scaling option can't be negotiated on the connection even if the <b>SO_RCVBUF</b> socket option was called on this socket specifying a value greater than 65,535 bytes before the connection was established.

The **TargetOsVersion** member can be one of the NTDDI version constants defined in the *Sdkddkver.h* header file. Some of the possible values for the **TargetOsVersion** member are as follows:

TERM	DESCRIPTION
NTDDI_LONGHORN	The target behavior is the default for Windows Vista.
NTDDI_WS03	The target behavior is the default for Windows Server 2003.
NTDDI_WINXP	The target behavior is the default for Windows XP.
NTDDI_WIN2K	The target behavior is the default for Windows 2000.

The primary impact of the **TargetOsVersion** member is whether this member is set to a value equal or greater than **NTDDI\_LONGHORN**.

TCP performance depends not only on the transfer rate itself, but rather on the product of the transfer rate and the round-trip delay time. This bandwidth-delay product measures the amount of data that would "fill the pipe". This bandwidth-delay product is the buffer space required at sender and receiver to obtain maximum throughput on the TCP connection over the path. This buffer space represents the amount of unacknowledged data that TCP must handle in order to keep the pipeline full. TCP performance problems arise when the bandwidth-delay product is large. A network path operating under these conditions is often called a "long, fat pipe". Examples include high-capacity packet satellite links, high-speed wireless links, and terrestrial fiber-optical links over long distances.

The TCP header uses a 16-bit data field (the Window field in the TCP packet header) to report the receive window size to the sender. Therefore, the largest window that can be used is 65,535 bytes. To circumvent this limitation a TCP extension option, TCP Window Scale, was added for high-performance TCP to allow windows larger than 65,535 bytes. The TCP Window Scale option (WSopt) is defined in RFC 1323 available at the [IETF](#)

[website](#). The WSopt extension expands the definition of the TCP window to 32 bits using a one-byte logarithmic scale factor to extend the 16-bit Window field in the TCP header. The WSopt extension defines an implicit scale factor (2 to some power), which is used to multiply the window size value found in a TCP header to obtain the true window size. So a window scale factor of 8 would result in a true window size equal to the value in the Window field in the TCP header multiplied by  $2^8$  or 256. So if the Window field in the TCP header was set to the maximum value of 65,535 bytes and the WSopt scale factor was negotiated to a value of 8, the true window size would be 16,776,960 bytes.

The true receive window size and therefore the scale factor is determined by the maximum receive buffer space. This maximum buffer space is the amount of data that a TCP receiver allows a TCP sender to send before having to wait for an acknowledgement. After the connection is established, the receive window size is advertised in each TCP segment (the Window field in the TCP header). Advertising the maximum amount of data that the sender can send is a receiver-side flow control mechanism that prevents the sender from sending data that the receiver cannot store. A sending host can only send the maximum amount of data advertised by the receiver before waiting for an acknowledgment and a receive window size update.

On Windows Server 2003 and Windows XP, the maximum receive buffer space which represents the receive window size for the TCP/IP stack has a default value based on the link speed of the sending interface. The actual value automatically adjusts to even increments of the maximum segment size (MSS) negotiated during TCP connection establishment. So for a 10 Mbit/sec link, the default receive window size would normally be set to 16K bytes, while on a 100 MBit/sec link the default receive window size would be set to 65,535 bytes.

On Windows Server 2003 and Windows XP, the true maximum receive window size for the TCP/IP stack can be manually configured using the following registry values on a specific interface or for the entire system:

`HKEY_LOCAL_MACHINE\SYSTEM\Current Control Set\Services\Tcpip\Parameters\TCPWindowSize`

`HKEY_LOCAL_MACHINE\SYSTEM\Current Control Set\Services\Tcpip\Parameters\Interface\TCPWindowSize`

The registry value for **TCPWindowSize** can be set to a maximum of 65,535 bytes when not using the WSopt extension or a maximum of 1,073,741,823 bytes when the WSopt extension is used (a maximum scale factor of 4 is supported). Without window scaling, an application can only achieve a throughput of approximately 5 megabits per second (Mbps) on a path with a 100 millisecond round-trip time (RTT), regardless of the path bandwidth. This throughput can be scaled to over a gigabit per second (Gbps) with window scaling, which allows TCP to negotiate the scaling factor for the window size during connection establishment.

On Windows Server 2003 and Windows XP, the WSopt extension can be enabled by setting the following registry value.

`HKEY_LOCAL_MACHINE\SYSTEM\Current Control Set\Services\Tcpip\Parameters\Tcp13230pts`

The **Tcp13230pts** registry value is a DWORD encoded such that when bit 0 is set, TCP WSopt extension is enabled. When bit 1 is set, the TCP Timestamp option (TSopt) defined in RFC 1323 is enabled. So a value of either 1 or 3 will enable the WSopt extension.

On Windows Server 2003 and Windows XP, the default is that the **TCPWindowSize** and the **Tcp13230pts** registry values are not created. So the default is that the WSopt extension is disabled and the TCP receive window size is set by the system to maximum value of up to 65,535 bytes based on the link speed. When window scaling is enabled on Windows Server 2003 and Windows XP by setting the **Tcp13230pts** registry value, window scaling on a TCP connection is still only used when both the sender and receiver include a TCP window scale option in the synchronize (SYN) segment sent to each other to negotiate a window scale factor. When window scaling is used on a connection, the Window field in the TCP header is set to 65,535 bytes and the window scale factor is used to adjust the true receive window size upward by the window scale factor negotiated when the connection is established.

An application can specify the TCP receive window size for a connection by using the **SO\_RCVBUF** socket option. The TCP receive window size for a socket can be increased at any time using **SO\_RCVBUF**, but it can

only be decreased prior to establishing a connection. In order to use window scaling, an application must specify a window size larger than 65,535 bytes when using the **SO\_RCVBUF** socket option before the connection is established.

The ideal value for the TCP receive window size is often difficult to determine. In order to fill the capacity of the network between the sender and receiver, the receive window size should be set to the bandwidth-delay product for the connection, which is the bandwidth multiplied by the round-trip time. Even if an application can correctly determine the bandwidth-delay product, it is still not known how quickly the receiving application will retrieve data from the incoming data buffer (the application retrieve rate). Despite the support for TCP window scaling, the maximum receive window size in Windows Server 2003 and Windows XP can still limit throughput because it is a fixed maximum size for all TCP connections (unless specified per application using **SO\_RCVBUF**), which can enhance throughput for some connections and decrease throughput for others. Additionally, the fixed maximum receive window size for a TCP connection does not vary with changing network conditions.

To solve the problem of correctly determining the value of the maximum receive window size for a TCP connection based on the current conditions of the network, the TCP/IP stack in Windows Vista supports a receive window auto-tuning feature. When this feature is enabled, receive window auto-tuning continually determines the optimal true receive window size by measuring the bandwidth-delay product and the application retrieve rate, and adjusts the true maximum receive window size based on changing network conditions. Receive window auto-tuning enables the TCP WSopt extension by default, allowing up to 16,776,960 bytes for the true window size. As the data flows over the connection, the TCP/IP stack monitors the connection, measures the current bandwidth-delay product for the connection and the application receive rate, and adjusts the actual receive window size to optimize throughput. The TCP/IP stack changes the value of the Window field in the TCP header based on network conditions, since the WSopt scale factor is fixed when the connection is first established.

The TCP/IP stack in Windows Vista no longer uses the **TCPWindowSize** registry values. With better throughput between TCP peers, the utilization of network bandwidth increases during data transfer. If all the applications are optimized to receive TCP data, then the overall utilization of the network can increase substantially, making the use of Quality of Service (QoS) more important on networks that are operating at or near capacity.

The default behavior on Windows Vista for receive buffering when **SIO\_SET\_COMPATIBILITY\_MODE** is not specified using **WsaBehaviorReceiveBuffering** is that no receive window size reductions using **SO\_RCVBUF** socket option are allowed after a connection is established.

The default behavior on Windows Vista for auto-tuning when **SIO\_SET\_COMPATIBILITY\_MODE** is not specified using **WsaBehaviorAutoTuning** is that the stack will do receive window auto-tuning using a window scale factor of 8. Note that if an application sets a valid receive window size with the **SO\_RCVBUF** socket option, the stack will use the size specified and window receive auto-tuning will be disabled. Windows autotuning may also be disabled completely using the following command,

```
netsh interface tcp set global autotuninglevel=disabled
```

 , in which case specifying **WsaBehaviorAutoTuning** will have no affect. Window receive autotuning can also be disabled based on group policy set on Windows Server 2008.

The **WsaBehaviorAutoTuning** option is needed on Windows Vista for some Internet gateway devices and firewalls that do not correctly support data flows for TCP connections that use the WSopt extension and a windows scale factor. On Windows Vista, a receiver by default negotiates a window scale factor of 8 for a maximum true window size of 16,776,960 bytes. When data begins to flow on a fast link, Windows initially starts with a 64 Kilobyte true window size by setting the Window field of the TCP header to 256 and setting the window scale factor to 8 in the TCP options ( $256 \times 2^8 = 64\text{KB}$ ). Some Internet gateway devices and firewalls ignore the window scale factor and only look at the advertised Window field in the TCP header specified as 256, and drop incoming packets for the connection that contain more than 256 bytes of TCP data. To support TCP receive window scaling, a gateway device or firewall must monitor the TCP handshake and track the negotiated window scale factor as part of the TCP connection data. Also some applications and TCP stack implementations on other platforms ignore the TCP WSopt extension and the window scaling factor. So the remote host sending

the data may send data at the rate advertised in the Window field of the TCP header (256 bytes). This can result in data being received very slowly by the receiver.

Setting the **BehaviorId** member to **WsaBehaviorAutoTuning** and the **TargetOsVersion** to Windows Vista reduces the window scale factor to 2, so the Window field in the TCP header is initially set to 16,384 bytes and the window scale factor is set to 2 for an initial true window receive size of 64K bytes. The window auto-tuning feature can then increase the true window receive size up to 262,140 bytes by setting the Window field in the TCP header to 65,535 bytes. An application should set the **SIO\_SET\_COMPATIBILITY\_MODE** IOCTL as soon as a socket is created, since this option doesn't make sense or apply after a SYN is sent. Setting this option has the same impact as the following command:

`netsh interface tcp set global autotuninglevel=highlyrestricted`

Note that the *Mswsockdef.h* header file is automatically included in *Mswsock.h* or *Netiodef.h*, and should not be used directly.

## See also

[socket](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_SET\_WFP\_CONNECTION\_REDIRECT\_RECORDS

## Control Code

3/5/2021 • 7 minutes to read • [Edit Online](#)

## Description

The SIO\_SET\_WFP\_CONNECTION\_REDIRECT\_RECORDS control code sets the redirect record to the new TCP socket used for connecting to the final destination for use by a Windows Filtering Platform (WFP) redirect service.

To perform this operation, call the [WSAIoctl](#) or [WSPIoctl](#) function with the following parameters.

```
int WSAIoctl(
    (socket) s,                                // descriptor identifying a socket
    SIO_SET_WFP_CONNECTION_REDIRECT_RECORDS, // dwIoControlCode
    (LPVOID) lpvInputBuffer,                    // lpvInBuffer
    (DWORD) cbInputBuffer,                     // cbInBuffer
    NULL,                                     // output buffer
    0,                                         // size of output buffer
    (LPDWORD) lpcbBytesReturned,               // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped,            // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);
```

```
int WSPIoctl(
    (socket) s,                                // descriptor identifying a socket
    SIO_SET_WFP_CONNECTION_REDIRECT_RECORDS, // dwIoControlCode
    (LPVOID) lpvInputBuffer,                    // lpvInBuffer
    (DWORD) cbInputBuffer,                     // cbInBuffer
    NULL,                                     // output buffer
    0,                                         // size of output buffer
    (LPDWORD) lpcbBytesReturned,               // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped,            // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId,                // a WSATHREADID structure
    (LPINT) lpErrno                           // a pointer to the error code.
);
```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use SIO\_SET\_WFP\_CONNECTION\_REDIRECT\_RECORDS for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter contains a pointer to the WFP redirect record associated with the socket.

### cbInBuffer

The size, in bytes, of the input buffer.

### **IpvOutBuffer**

A pointer to the output buffer. This parameter is unused for this operation.

### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter must be set to zero.

### **IpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer.

If the output buffer is too small, the call fails, [WSAGetLastError](#) returns [WSAEINVAL](#), and the *IpcbBytesReturned* parameter points to a **DWORD** value of zero.

If *lpOverlapped* is **NULL**, the **DWORD** value pointed to by the *IpcbBytesReturned* parameter that is returned on a successful call cannot be zero.

If the *lpOverlapped* parameter is not **NULL** for overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The **DWORD** value pointed to by the *IpcbBytesReturned* parameter that is returned may be zero since the size of the data stored can't be determined until the overlapped operation has completed. The final completion status can be retrieved when the appropriate completion method is signaled when the operation has completed.

### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAIoctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

### **IpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

### **IpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

### **IpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAIoctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAIoctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
<b>WSA_IO_PENDING</b>	Overlapped I/O operation is in progress. This value is returned if an overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSA_OPERATION_ABORTED</b>	The I/O operation has been aborted because of either a thread exit or an application request. This error is returned if an overlapped operation was canceled due to the closure of the socket or the execution of the <b>SIO_FLUSH</b> IOCTL command.
<b>WSAEACCES</b>	An attempt was made to access a socket in a way forbidden by its access permissions. This error is returned under several conditions that include the following: the user lacks the required administrative privileges on the local computer or the application is not running in an enhanced shell as the built-in Administrator ( <a href="#">RunAs administrator</a> ).
<b>WSAEFAULT</b>	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if the <i>lpvInBuffer</i> , <i>lpvoutBuffer</i> , <i>lpcbBytesReturned</i> , <i>lpOverlapped</i> or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.
<b>WSAEINPROGRESS</b>	A blocking operation is currently executing. This error is returned if the function is invoked when a callback is in progress.
<b>WSAEINTR</b>	A blocking operation was interrupted by a call to <b>WSACancelBlockingCall</b> . This error is returned if a blocking operation was interrupted.
<b>WSAEINVAL</b>	An invalid argument was supplied. This error is returned if the <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified.
<b>WSAENETDOWN</b>	A socket operation encountered a dead network. This error is returned if the network subsystem has failed.
<b>WSAENOTSOCK</b>	An operation was attempted on something that is not a socket. This error is returned if the descriptor <i>s</i> is not a socket.
<b>WSAEOPNOTSUPP</b>	The attempted operation is not supported for the type of object referenced. This error is returned if the specified IOCTL command is not supported. This error is also returned if the <b>SIO_SET_WFP_CONNECTION_REDIRECT_RECORDS</b> IOCTL is not supported by the transport provider.

## Remarks

The **SIO\_SET\_WFP\_CONNECTION\_REDIRECT\_RECORDS** IOCTL is supported on Windows 8, and Windows

Server 2012, and later versions of the operating system.

WFP allows access to the TCP/IP packet processing path, wherein outgoing and incoming packets can be examined or changed before allowing them to be processed further. By tapping into the TCP/IP processing path, independent software vendors (ISVs) can more easily create firewalls, antivirus software, diagnostic software, and other types of applications and services. WFP provides user-mode and kernel-mode components so that third-party ISVs can participate in the filtering decisions that take place at several layers in the TCP/IP protocol stack and throughout the operating system. The WFP connection redirect feature allows a WFP callout kernel driver to redirect a connection locally to a user-mode process, perform content inspection in user-mode, and forward the inspected content using a different connection to the original destination.

The [SIO\\_SET\\_WFP\\_CONNECTION\\_REDIRECT\\_RECORDS](#) IOCTL and several other related IOCTLS are user-mode components used to allow multiple WFP-based connection proxy applications to inspect the same traffic flow in a cooperative manner. Each inspection agent can safely re-inspect network traffic that has already been inspected by another inspection agent. With the presence of multiple proxies (developed by different ISVs, for example) the connection used by one proxy to communicate with the final destination could in turn be redirected by a second proxy, and that new connection could again be redirected by the original proxy. Without connection tracking, the original connection might never reach its final destination as it gets stuck in an infinite proxy loop.

The [SIO\\_SET\\_WFP\\_CONNECTION\\_REDIRECT\\_RECORDS](#) IOCTL is used to provide proxied connection tracking on redirected socket connections. This WFP feature facilitates tracking of redirection records from the initial redirect of a connection to the final connection to the destination.

The [SIO\\_QUERY\\_WFP\\_CONNECTION\\_REDIRECT\\_RECORDS](#) IOCTL is used by a WFP-based redirect service to retrieve the redirect record from the accepted TCP/IP packet connection (the connected socket for a TCP socket or a UDP socket, for example) redirected to it by its companion kernel-mode callout registered at [ALE\\_CONNECT\\_REDIRECT](#) layers in a kernel-mode driver. The [SIO\\_QUERY\\_WFP\\_CONNECTION\\_REDIRECT\\_CONTEXT](#) IOCTL retrieves the redirect context for a redirect record, which is used. The redirect context is optional and is used if the current redirection state of a connection is that the connection was redirected by the calling redirect service or the connection was previously redirected by the calling redirect service but later redirected again by a different redirect service. The redirect service transfers the retrieved redirect record to the TCP socket it uses to proxy the original content using the [SIO\\_SET\\_WFP\\_CONNECTION\\_REDIRECT\\_RECORDS](#) IOCTL.

The application calling the [SIO\\_SET\\_WFP\\_CONNECTION\\_REDIRECT\\_RECORDS](#) IOCTL does not need to understand the blob containing the redirect record being set. This is an opaque blob of data that the application needs to pass back to the new socket.

## See also

[IPPROTO\\_IP](#) Socket Options

[SIO\\_QUERY\\_WFP\\_CONNECTION\\_REDIRECT\\_CONTEXT](#)

[SIO\\_QUERY\\_WFP\\_CONNECTION\\_REDIRECT\\_RECORDS](#)

[socket](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_TCP\_INFO Control Code

3/6/2021 • 3 minutes to read • [Edit Online](#)

## Description

The SIO\_TCP\_INFO control code retrieves the Transmission Control Protocol (TCP) statistics for a specified socket.

To perform this operation, call the [WSAIoctl](#) or [WSPIoctl](#) function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_TCP_INFO,        // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to a DWORD
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    (LPVOID) lpvOutBuffer, // pointer to a TCP_INFO_v0 structure
    (DWORD) cbOutBuffer,  // size of the output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);

```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_TCP_INFO,        // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to a DWORD
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    (LPVOID) lpvOutBuffer, // pointer to a TCP_INFO_v0 structure
    (DWORD) cbOutBuffer,  // size of the output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);

```

## Parameters

### s

A descriptor that identifies a socket.

### dwIoControlCode

The control code for the operation. Use SIO\_TCP\_INFO for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter contains a pointer to a DWORD that specifies the version of the SIO\_TCP\_INFO control code that you are using. Specify 0 to use [TCP\\_INFO\\_v0](#). Specify 1 to use [TCP\\_INFO\\_v1](#), which provides more fields.

### cbInBuffer

The size, in bytes, of the input buffer. This parameter should be the size of the DWORD data type.

### lpvOutBuffer

A pointer to the output buffer. On successful output, this parameter contains a pointer to a [TCP\\_INFO\\_v0](#) structure that contains the TCP statistics for the specified socket.

#### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter must be at least the size of the [TCP\\_INFO\\_v0](#) structure.

#### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer.

If the output buffer is too small, the call fails, [WSAGetLastError](#) returns [WSAEINVAL](#), and the *lpcbBytesReturned* parameter points to a **DWORD** value of zero.

If *lpOverlapped* is **NULL**, the **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned on a successful call cannot be zero.

If the *lpOverlapped* parameter is not **NULL** for overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned may be zero since the size of the data stored can't be determined until the overlapped operation has completed. The final completion status can be retrieved when the appropriate completion method is signaled when the operation has completed.

#### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAOctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

#### **lpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

#### **lpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#).

The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

#### **lpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAOctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAOctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
<b>WSAEMSGSIZE</b>	The pointer to the input buffer was <b>NULL</b> , or the specified size of the input buffer was not correct.
<b>WSAEINVAL</b>	An invalid argument was supplied. This error is returned if the <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified.

## Remarks

Unlike retrieving TCP statistics with the [GetPerTcpConnectionEStats](#) function, retrieving TCP statistics with this control code does not require the user code to load, store, and filter the TCP connection table, and does not require elevated privileges to use.

## See also

[socket](#)

[TCP\\_INFO\\_v0](#)

[GetPerTcpConnectionEStats](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# SIO\_TCP\_INITIAL\_RTO control code

11/2/2020 • 5 minutes to read • [Edit Online](#)

## Description

The SIO\_TCP\_INITIAL\_RTO control code configures initial retransmission timeout (RTO) parameters on a socket.

To perform this operation, call the [WSAIoctl](#) or [WSPIoctl](#) function with the following parameters.

```
int WSAIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_TCP_INITIAL_RTO, // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to a TCP_INITIAL_RTO_PARAMETERS structure
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    (LPVOID) lpvOutBuffer, // pointer to output buffer
    (DWORD) cbOutBuffer,  // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
);

```

```
int WSPIoctl(
    (socket) s,           // descriptor identifying a socket
    SIO_TCP_INITIAL_RTO, // dwIoControlCode
    (LPVOID) lpvInBuffer, // pointer to a TCP_INITIAL_RTO_PARAMETERS structure
    (DWORD) cbInBuffer,   // size, in bytes, of the input buffer
    (LPVOID) lpvOutBuffer, // pointer to output buffer
    (DWORD) cbOutBuffer,  // size of output buffer
    (LPDWORD) lpcbBytesReturned, // number of bytes returned
    (LPWSAOVERLAPPED) lpOverlapped, // OVERLAPPED structure
    (LPWSAOVERLAPPED_COMPLETION_ROUTINE) lpCompletionRoutine, // completion routine
    (LPWSATHREADID) lpThreadId, // a WSATHREADID structure
    (LPINT) lpErrno // a pointer to the error code.
);

```

## Parameters

### s

A descriptor identifying a socket.

### dwIoControlCode

The control code for the operation. Use SIO\_TCP\_INITIAL\_RTO for this operation.

### lpvInBuffer

A pointer to the input buffer. This parameter contains a pointer to the [TCP\\_INITIAL\\_RTO\\_PARAMETERS](#) associated with the socket.

### cbInBuffer

The size, in bytes, of the input buffer.

### lpvOutBuffer

A pointer to the output buffer. This parameter is unused for this operation.

### **cbOutBuffer**

The size, in bytes, of the output buffer. This parameter must be set to zero.

### **lpcbBytesReturned**

A pointer to a variable that receives the size, in bytes, of the data stored in the output buffer.

If the output buffer is too small, the call fails, [WSAGetLastError](#) returns [WSAEINVAL](#), and the *lpcbBytesReturned* parameter points to a **DWORD** value of zero.

If *lpOverlapped* is **NULL**, the **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned on a successful call cannot be zero.

If the *lpOverlapped* parameter is not **NULL** for overlapped sockets, operations that cannot be completed immediately will be initiated, and completion will be indicated at a later time. The **DWORD** value pointed to by the *lpcbBytesReturned* parameter that is returned may be zero since the size of the data stored can't be determined until the overlapped operation has completed. The final completion status can be retrieved when the appropriate completion method is signaled when the operation has completed.

### **IpvOverlapped**

A pointer to a [WSAOVERLAPPED](#) structure.

If socket *s* was created without the overlapped attribute, the *lpOverlapped* parameter is ignored.

If *s* was opened with the overlapped attribute and the *lpOverlapped* parameter is not **NULL**, the operation is performed as an overlapped (asynchronous) operation. In this case, *lpOverlapped* parameter must point to a valid [WSAOVERLAPPED](#) structure.

For overlapped operations, the [WSAOIoctl](#) or [WSPIoctl](#) function returns immediately, and the appropriate completion method is signaled when the operation has been completed. Otherwise, the function does not return until the operation has been completed or an error occurs.

### **lpCompletionRoutine**

Type: `_In_opt_ LPWSAOVERLAPPED_COMPLETION_ROUTINE`

A pointer to the completion routine called when the operation has been completed (ignored for non-overlapped sockets).

### **lpThreadId**

A pointer to a [WSATHREADID](#) structure to be used by the provider in a subsequent call to [WPUQueueApc](#). The provider should store the referenced [WSATHREADID](#) structure (not the pointer to same) until after the [WPUQueueApc](#) function returns.

**Note** This parameter applies only to the [WSPIoctl](#) function.

### **lpErrno**

A pointer to the error code.

**Note** This parameter applies only to the [WSPIoctl](#) function.

## Return value

If the operation completes successfully, the [WSAOIoctl](#) or [WSPIoctl](#) function returns zero.

If the operation fails or is pending, the [WSAOIoctl](#) or [WSPIoctl](#) function returns **SOCKET\_ERROR**. To get extended error information, call [WSAGetLastError](#).

ERROR CODE	MEANING
<b>WSA_IO_PENDING</b>	Overlapped I/O operation is in progress. This value is returned if an overlapped operation was successfully initiated and completion will be indicated at a later time.
<b>WSA_OPERATION_ABORTED</b>	The I/O operation has been aborted because of either a thread exit or an application request. This error is returned if an overlapped operation was canceled due to the closure of the socket or the execution of the <b>SIO_FLUSH</b> IOCTL command.
<b>WSAEACCES</b>	An attempt was made to access a socket in a way forbidden by its access permissions. This error is returned under several conditions that include the following: the user lacks the required administrative privileges on the local computer or the application is not running in an enhanced shell as the built-in Administrator ( <code>RunAs administrator</code> ).
<b>WSAEFAULT</b>	The system detected an invalid pointer address in attempting to use a pointer argument in a call. This error is returned if the <i>lpvInBuffer</i> , <i>lpvoutBuffer</i> , <i>lpcbBytesReturned</i> , <i>lpOverlapped</i> or <i>lpCompletionRoutine</i> parameter is not totally contained in a valid part of the user address space.
<b>WSAEINPROGRESS</b>	A blocking operation is currently executing. This error is returned if the function is invoked when a callback is in progress.
<b>WSAEINTR</b>	A blocking operation was interrupted by a call to <i>WSACancelBlockingCall</i> . This error is returned if a blocking operation was interrupted.
<b>WSAEINVAL</b>	An invalid argument was supplied. This error is returned if the <i>dwIoControlCode</i> parameter is not a valid command, or a specified input parameter is not acceptable, or the command is not applicable to the type of socket specified.
<b>WSAENETDOWN</b>	A socket operation encountered a dead network. This error is returned if the network subsystem has failed.
<b>WSAENOTSOCK</b>	An operation was attempted on something that is not a socket. This error is returned if the descriptor <i>s</i> is not a socket.
<b>WSAEOPNOTSUPP</b>	The attempted operation is not supported for the type of object referenced. This error is returned if the specified IOCTL command is not supported. This error is also returned if the <b>SIO_TCP_INITIAL_RTO</b> IOCTL is not supported by the transport provider.

## Remarks

An application can use the **SIO\_TCP\_INITIAL\_RTO** IOCTL to control the initial (SYN / SYN+ACK) retransmission characteristics of a TCP socket if required. An application, if using this option, must supply suitable values before starting a TCP connection attempt on the socket.

The **TCP\_INITIAL\_RTO\_PARAMETERS** IOCTL allows an application to configure the initial (SYN)

retransmission timeout (RTO) used by the socket.

A pointer to a [TCP\\_INITIAL\\_RTO\\_PARAMETERS](#) structure passed in the *lpvInBuffer* parameter allows an application to configure the initial round trip time (RTT) used to compute the retransmission timeout. The application can also configure the number of retransmissions that will be attempted before the connection attempt fails.

## See also

[socket](#)

[TCP\\_INITIAL\\_RTO\\_PARAMETERS](#)

[WSAGetLastError](#)

[WSAGetOverlappedResult](#)

[WSAIoctl](#)

[WSAOVERLAPPED](#)

[WSASocketA](#)

[WSASocketW](#)

# Winsock Annexes

3/5/2021 • 2 minutes to read • [Edit Online](#)

Winsock Annexes provide implementation information for common Winsock protocol suites, and discuss how to use each protocol with Winsock.

Each protocol in this annex has conventions, behaviors, or special features that do not lend themselves to generic implementation. This section documents details that developers must consider when implementing or using the described protocols.

The Winsock Annexes include the following sections:

- [Winsock ATM Annex](#)
- [Winsock IPX/SPX Annex](#)
- [Winsock TCP/IP Annex](#)

# Winsock ATM Annex

3/5/2021 • 2 minutes to read • [Edit Online](#)

ATM is applicable to both LAN and WAN environments. An ATM network simultaneously transports a wide variety of network traffic — voice, data, image, and video. It provides users with a guaranteed quality of service on a per-virtual channel (VC) basis.

ELEMENT	DESCRIPTION
Protocol name(s)	ATMPROTO_AAL5, ATMPROTO_AALUSER
Description	ATM AAL5 provides a transport service that is connection oriented, message-boundary preserved, and QOS guaranteed. ATMPROTO_AALUSER is a user-defined AAL.
Address family	AF_ATM
Header file	Ws2atm.h

This section describes the Asynchronous Transfer Mode (ATM)-specific extensions needed to support the native ATM services as exposed and specified in the ATM Forum User Network Interface (UNI) specification version 3.x (3.0 and 3.1). This document supports AAL type 5 (message mode) and user-defined AAL. Future versions of this document will support other types of AAL as well as UNI 4.0.

# Winsock ATM Controls

3/5/2021 • 2 minutes to read • [Edit Online](#)

ATM point-to-point and point-to-multipoint connection setup and teardown are natively supported by the Windows Sockets 2 specification. In fact, Windows Sockets 2 QoS specification and protocol-independent multipoint/multicast mechanisms were designed with ATM in mind, along with other protocols. See Section 2.7 and Appendix D of the Windows Sockets 2 API specification for Windows Sockets 2, Quality of Service, and Multipoint Support, respectively. Therefore, no ATM-specific loctls need to be introduced in this document.

# Winsock ATM Function Details

3/5/2021 • 2 minutes to read • [Edit Online](#)

Based on the taxonomy defined for Windows Sockets 2 protocol-independent multipoint/multicast schemes, ATM falls into the category of rooted data and rooted control planes. (See the Windows Sockets 2 API Specification, Appendix D for more information.) 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 the listening mode. [WSAJoinLeaf](#) with a c\_root socket specified will be mapped to the Q.2931 SETUP message (for the first leaf) or ADD PARTY message (for any subsequent leaves).

## NOTE

The QoS parameters specified in [WSAJoinLeaf](#) for any subsequent leaves should be ignored per the ATM Forum UNI specification.

The leaf-initiated join is not part of the ATM UNI 3.1, but is supported in the ATM UNI 4.0. Thus [WSAJoinLeaf](#) with a c\_leaf socket specified can be mapped to the appropriate ATM UNI 4.0 message.

The AAL Parameter and B-LLI negotiation is supported through the modification of the corresponding IEs in the */pSQOS* parameter upon the invocation of the condition function specified in [WSAAccept](#).

## NOTE

Only certain fields in those two IEs can be modified. See the ATM Forum UNI specification Appendix C and Appendix F for details.

# Winsock ATM QoS Extension

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section describes the protocol-specific Quality of Service (**QoS**) structure for ATM, which is contained in the **ProviderSpecific** field of the **QOS** structure. Note that the use of this ATM-specific **QoS** structure is optional by Windows Sockets 2 clients, and the ATM service provider is required to map the generic **FLOWSPEC** structure to appropriate Q.2931 information elements. However, if both the generic **FLOWSPEC** structure and ATM-specific **QoS** structure are specified, the value specified in the ATM-specific **QoS** structure should take precedence should there be any conflicts. See Windows Sockets 2 API Specification Section 2.7 for more information about QoS provisions and **FLOWSPEC** structure.

The protocol-specific **QoS** structure for ATM is a concatenation of Q.2931 information element (IE) structures, which are defined in the following text. If an application omits an IE that UNI 3.x mandates, the service provider should insert a reasonable default value, taking the information in the **FLOWSPEC** structure into account if applicable.

The handling of repeated IEs is dependent on the IE itself. If an IE is repeated and it is one that is allowed to be repeated per the ATM Forum UNI specification then the provider must handle it properly. In this case, order in the list determines preference order, with elements appearing earlier in the list being more preferred. If an IE is repeated and this is not allowed per ATM Forum UNI specification, the provider may either fail the request from the client (the preferred option) or use the last IE of that type found.

Each individual IE structure is formatted in the following manner and identified by the **IEType** field:

```
typedef struct {
    Q2931_IE_TYPE IEType;
    ULONG         IELength;
    UCHAR         IE[1];
} Q2931_IE;
```

Legal values for the **IEType** field are defined as:

```
typedef enum {
    IE_AALParameters,
    IE_TrafficDescriptor,
    IE_BroadbandBearerCapability,
    IE_BHLI,
    IE_BLLI,
    IE_CalledPartyNumber,
    IE_CalledPartySubaddress,
    IE_CallingPartyNumber,
    IE_CallingPartySubaddress,
    IE_Cause,
    IE_QOSClass,
    IE_TransitNetworkSelection,
} Q2931_IE_TYPE;
```

The **IE** field is overlaid by a specific IE structure and the **IELength** field is the total length in bytes of the IE structure including the **IEType** and **IELength** fields. The semantics and legal values for each element of these IE structures are per ATM UNI 3.x specification. **SAP\_FIELD\_ABSENT** can be used for those elements that are optional for a given IE structure, per ATM UNI 3.x specification.



# AAL Parameters

3/5/2021 • 2 minutes to read • [Edit Online](#)

```
#include <windows.h>

/*
 * manifest constants for the AALType field in struct AAL_PARAMETERS_IE
 */
typedef enum {
    AALTYPE_5      = 5,    /* AAL 5 */
    AALTYPE_USER   = 16    /* user-defined AAL */
} AAL_TYPE;

/*
 * values used for the Mode field in struct AAL5_PARAMETERS
 */
#define AAL5_MODE_MESSAGE        0x01
#define AAL5_MODE_STREAMING      0x02

/*
 * values used for the SSCSType field in struct AAL5_PARAMETERS
 */
#define AAL5_SSCS_NULL           0x00
#define AAL5_SSCS_SSCOP_ASSURED  0x01
#define AAL5_SSCS_SSCOP_NON_ASSURED 0x02
#define AAL5_SSCS_FRAME_RELAY    0x04

typedef struct {
    ULONG ForwardMaxCPCSSDUSize;
    ULONG BackwardMaxCPCSSDUSize;
    UCHAR Mode;                      /* only available in UNI 3.0 */
    UCHAR SSCSType;
} AAL5_PARAMETERS;

typedef struct {
    ULONG UserDefined;
} AALUSER_PARAMETERS;

typedef struct {
    AAL_TYPE AALType;
    union {
        AAL5_PARAMETERS     AAL5Parameters;
        AALUSER_PARAMETERS  AALUserParameters;
    } AALSspecificParameters;
} AAL_PARAMETERS_IE;
```

# ATM Traffic Descriptor

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section lists the ATM traffic descriptor.

```
typedef struct {
    ULONG PeakCellRate_CLP0;
    ULONG PeakCellRate_CLP01;
    ULONG SustainableCellRate_CLP0;
    ULONG SustainableCellRate_CLP01;
    ULONG MaxBurstSize_CLP0;
    ULONG MaxBurstSize_CLP01;
    BOOL Tagging;
} ATM_TD;

typedef struct {
    ATM_TD Forward;
    ATM_TD Backward;
    BOOL BestEffort;
} ATM_TRAFFIC_DESCRIPTOR_IE;
```

# Broadband Bearer Capability

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section lists the values used for the broadband bearer capability.

```
#include <windows.h>

/*
 *  values used for the BearerClass field in struct ATM_BROADBAND_BEARER_CAPABILITY_IE
 */
#define BCOB_A          0x00 /* Bearer class A      */
#define BCOB_C          0x03 /* Bearer class C      */
#define BCOB_X          0x10 /* Bearer class X      */

/*
 *  values used for the TrafficType field in struct ATM_BROADBAND_BEARER_CAPABILITY_IE
 */
#define TT_NOIND        0x00 /* No indication of traffic type      */
#define TT_CBR          0x04 /* Constant bit rate      */
#define TT_VBR          0x06 /* Variable bit rate      */

/*
 *  values used for the TimingRequirements field in struct ATM_BROADBAND_BEARER_CAPABILITY_IE
 */
#define TR_NOIND        0x00 /* No timing requirement indication      */
#define TR_END_TO_END    0x01 /* End-to-end timing required      */
#define TR_NO_END_TO_END 0x02 /* End-to-end timing not required      */

/*
 *  values used for the ClippingSusceptability field in struct ATM_BROADBAND_BEARER_CAPABILITY_IE
 */
#define CLIP_NOT        0x00 /* Not susceptible to clipping      */
#define CLIP_SUS         0x20 /* Susceptible to clipping      */

/*
 *  values used for the UserPlaneConnectionConfig field in
 *  struct ATM_BROADBAND_BEARER_CAPABILITY_IE
 */
#define UP_P2P           0x00 /* Point-to-point connection      */
#define UP_P2MP          0x01 /* Point-to-multipoint connection      */

typedef struct {
    UCHAR BearerClass;
    UCHAR TrafficType;
    UCHAR TimingRequirements;
    UCHAR ClippingSusceptability;
    UCHAR UserPlaneConnectionConfig;
} ATM_BROADBAND_BEARER_CAPABILITY_IE;
```

# Broadband High Layer Information

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section lists the type definition for the broadband high-layer information.

```
typedef ATM_BHLI ATM_BHLI_IE;
```

# Broadband Lower Layer Information

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section lists the type definition for the broadband lower-layer information.

```
#include <windows.h>

/*
 * values used for the Layer2Mode field in struct ATM_BLLI_IE
 */
#define BLLI_L2_MODE_NORMAL      0x40
#define BLLI_L2_MODE_EXT         0x80

/*
 * values used for the Layer3Mode field in struct ATM_BLLI_IE
 */
#define BLLI_L3_MODE_NORMAL      0x40
#define BLLI_L3_MODE_EXT         0x80

/*
 * values used for the Layer3DefaultPacketSize field in struct ATM_BLLI_IE
 */
#define BLLI_L3_PACKET_16          0x04
#define BLLI_L3_PACKET_32          0x05
#define BLLI_L3_PACKET_64          0x06
#define BLLI_L3_PACKET_128         0x07
#define BLLI_L3_PACKET_256         0x08
#define BLLI_L3_PACKET_512         0x09
#define BLLI_L3_PACKET_1024        0x0A
#define BLLI_L3_PACKET_2048        0x0B
#define BLLI_L3_PACKET_4096        0x0C

typedef struct {
    DWORD Layer2Protocol;           /* User information layer 2 protocol */
    UCHAR Layer2Mode;
    UCHAR Layer2WindowSize;
    DWORD Layer2UserSpecifiedProtocol; /* User specified layer 2 protocol information */
    DWORD Layer3Protocol;           /* User information layer 3 protocol */
    UCHAR Layer3Mode;
    UCHAR Layer3DefaultPacketSize;
    UCHAR Layer3PacketWindowSize;
    DWORD Layer3UserSpecifiedProtocol; /* User specified layer 3 protocol information */
    DWORD Layer3IPI;                /* ISO/IEC TR 9577 Initial Protocol Identifier */
    UCHAR SnapID[5];                /* SNAP ID consisting of OUI and PID */
} ATM_BLLI_IE;
```

# Called Party Number

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section lists the type definition for the called party number.

```
typedef ATM_ADDRESS ATM_CALLED_PARTY_NUMBER_IE;
```

# Called Party Subaddress

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section lists the type definition for the called party subaddress.

```
typedef ATM_ADDRESS ATM_CALLED_PARTY_SUBADDRESS_IE;
```

# Calling Party Number

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section lists the type definition for the calling party number.

```
#include <windows.h>

/*
 *  values used for the Presentation_Indication field in
 *  struct ATM_CALLING_PARTY_NUMBER_IE
 */
#define PI_ALLOWED          0x00
#define PI_RESTRICTED       0x40
#define PI_NUMBER_NOT_AVAILABLE 0x80

/*
 *  values used for the Screening_Indicator field in
 *  struct ATM_CALLING_PARTY_NUMBER_IE
 */
#define SI_USER_NOT_SCREENED 0x00
#define SI_USER_PASSED      0x01
#define SI_USER_FAILED       0x02
#define SI_NETWORK           0x03

typedef struct {
    ATM_ADDRESS ATM_Number;
    UCHAR     Presentation_Indication;
    UCHAR     Screening_Indicator;
} ATM_CALLING_PARTY_NUMBER_IE;
```

# Calling Party Subaddress

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section lists the type definition for the calling party subaddress.

```
typedef ATM_ADDRESS ATM_CALLED_PARTY_NUMBER_IE;
```

# Quality of Service Parameter

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section lists the parameters used for quality of service (QoS).

```
#include <windows.h>
/*
 *  values used for the QOSClassForward and QOSClassBackward
 *  field in struct ATM_QOS_CLASS_IE
 */
#define QOS_CLASS0          0x00
#define QOS_CLASS1          0x01
#define QOS_CLASS2          0x02
#define QOS_CLASS3          0x03
#define QOS_CLASS4          0x04

typedef struct {
    UCHAR QOSClassForward;
    UCHAR QOSClassBackward;
} ATM_QOS_CLASS_IE;
```

# Transit Network Selection

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section lists values used for the transit network selection.

```
#include <windows.h>
/*
 *  values used for the TypeOfNetworkId field in struct ATM_TRANSIT_NETWORK_SELECTION_IE
 */
#define TNS_TYPE_NATIONAL          0x40

/*
 *  values used for the NetworkIdPlan field in struct ATM_TRANSIT_NETWORK_SELECTION_IE
 */
#define TNS_PLAN_CARRIER_ID_CODE   0x01

typedef struct {
    UCHAR TypeOfNetworkId;
    UCHAR NetworkIdPlan;
    UCHAR NetworkIdLength;
    UCHAR NetworkId[1];
} ATM_TRANSIT_NETWORK_SELECTION_IE;
```

# Cause

3/5/2021 • 2 minutes to read • [Edit Online](#)

In addition to all the information elements previously described, which could be specified in the ATM-specific **QoS** structure while calling **WSAConnect**, there is a Cause IE that can only be used during the call release. Upon disconnecting, Windows Sockets 2 applications can optionally specify this IE as the disconnect data in **WSASendDisconnect**. The remote party can retrieve this IE through **WSARecvDisconnect** after receiving the FD\_CLOSE notification.

```
#include <windows.h>

/*
 * values used for the Location field in struct ATM_CAUSE_IE
 */
#define CAUSE_LOC_USER          0x00
#define CAUSE_LOC_PRIVATE_LOCAL  0x01
#define CAUSE_LOC_PUBLIC_LOCAL   0x02
#define CAUSE_LOC_TRANSIT_NETWORK 0x03
#define CAUSE_LOC_PUBLIC_REMOTE  0x04
#define CAUSE_LOC_PRIVATE_REMOTE 0x05
#define CAUSE_LOC_INTERNATIONAL_NETWORK 0x06
#define CAUSE_LOC_BEYOND_INTERWORKING 0x0A

/*
 * values used for the Cause field in struct ATM_CAUSE_IE
 */
#define CAUSE_UNALLOCATED_NUMBER 0x01
#define CAUSE_NO_ROUTE_TO_TRANSIT_NETWORK 0x02
#define CAUSE_NO_ROUTE_TO_DESTINATION 0x03
#define CAUSE_VPI_VCI_UNACCEPTABLE 0x0A
#define CAUSE_NORMAL_CALL_CLEARING 0x10
#define CAUSE_USER_BUSY          0x11
#define CAUSE_NO_USER_RESPONDING 0x12
#define CAUSE_CALL_REJECTED      0x15
#define CAUSE_NUMBER_CHANGED     0x16
#define CAUSE_USER_REJECTS_CLIR  0x17
#define CAUSE_DESTINATION_OUT_OF_ORDER 0x1B
#define CAUSE_INVALID_NUMBER_FORMAT 0x1C
#define CAUSE_STATUS_ENQUIRY_RESPONSE 0x1E
#define CAUSE_NORMAL_UNSPECIFIED 0x1F
#define CAUSE_VPI_VCI_UNAVAILABLE 0x23
#define CAUSE_NETWORK_OUT_OF_ORDER 0x26
#define CAUSE_TEMPORARY_FAILURE 0x29
#define CAUSE_ACCESS_INFORMAION_DISCARDED 0x2B
#define CAUSE_NO_VPI_VCI_AVAILABLE 0x2D
#define CAUSE_RESOURCE_UNAVAILABLE 0x2F
#define CAUSE_QOS_UNAVAILABLE    0x31
#define CAUSE_USER_CELL_RATE_UNAVAILABLE 0x33
#define CAUSE_BEARER_CAPABILITY_UNAUTHORIZED 0x39
#define CAUSE_BEARER_CAPABILITY_UNAVAILABLE 0x3A
#define CAUSE_OPTION_UNAVAILABLE 0x3F
#define CAUSE_BEARER_CAPABILITY_UNIMPLEMENTED 0x41
#define CAUSE_UNSUPPORTED_TRAFFIC_PARAMETERS 0x49
#define CAUSE_INVALID_CALL_REFERENCE 0x51
#define CAUSE_CHANNEL_NONEXISTENT 0x52
#define CAUSE_INCOMPATIBLE_DESTINATION 0x58
#define CAUSE_INVALID_ENDPOINT_REFERENCE 0x59
#define CAUSE_INVALID_TRANSIT_NETWORK_SELECTION 0x5B
#define CAUSE_TOO_MANY_PENDING_ADD_PARTY 0x5C
#define CAUSE_AAL_PARAMETERS_UNSUPPORTED 0x5D
#define CAUSE_MANDATORY_IE_MISSING 0x60
```

```

#define CAUSE_UNIMPLEMENTED_MESSAGE_TYPE      0x61
#define CAUSE_UNIMPLEMENTED_IE                0x63
#define CAUSE_INVALID_IE_CONTENTS            0x64
#define CAUSE_INVALID_STATE_FOR_MESSAGE      0x65
#define CAUSE_RECOVERY_ON_TIMEOUT           0x66
#define CAUSE_INCORRECT_MESSAGE_LENGTH       0x68
#define CAUSE_PROTOCOL_ERROR                 0x6F

/*
 *  values used for the Condition portion of the Diagnostics field
 *  in struct ATM_CAUSE_IE, for certain Cause values
 */
#define CAUSE_COND_UNKNOWN                  0x00
#define CAUSE_COND_PERMANENT                0x01
#define CAUSE_COND_TRANSIENT               0x02

/*
 *  values used for the Rejection Reason portion of the Diagnostics field
 *  in struct ATM_CAUSE_IE, for certain Cause values
 */
#define CAUSE_REASON_USER                  0x00
#define CAUSE_REASON_IE_MISSING             0x04
#define CAUSE_REASON_IE_INSUFFICIENT       0x08

/*
 *  values used for the P-U flag of the Diagnostics field
 *  in struct ATM_CAUSE_IE, for certain Cause values
 */
#define CAUSE_PU_PROVIDER                 0x00
#define CAUSE_PU_USER                     0x08

/*
 *  values used for the N-A flag of the Diagnostics field
 *  in struct ATM_CAUSE_IE, for certain Cause values
 */
#define CAUSE_NA_NORMAL                   0x00
#define CAUSE_NA_ABNORMAL                 0x04

typedef struct {
    UCHAR Location;
    UCHAR Cause;
    UCHAR DiagnosticsLength;
    UCHAR Diagnostics[4];
} ATM_CAUSE_IE;

```

# Winsock ATM Structures

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following list provides concise descriptions of each Winsock ATM structure. For additional information on any structure, click the structure name.

ATM STRUCTURE	DESCRIPTION
<a href="#">ATM_ADDRESS</a>	Stores ATM address data for ATM-based sockets.
<a href="#">ATM_BHLI</a>	Identifies B-HLI information for an associated ATM socket.
<a href="#">ATM_BLLI</a>	Identifies B-LLI information for an associated ATM socket.
<a href="#">sockaddr_atm</a>	Stores socket address information for ATM sockets.

For native ATM services, use AF\_ATM for address family and the [sockaddr\\_atm](#) socket address structure. To open a native ATM socket, pass AF\_ATM, SOCK\_RAW, and ATMPROTO\_AAL5 or ATMPROTO\_AALUSER into the [socket](#) function, respectively.

# Winsock IPX/SPX Annex

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section describes Winsock extensions specific to Internetwork Packet Exchange/Sequenced Packet Exchange (IPX/SPX). It also describes aspects of base Winsock functions that either require special consideration or may exhibit unique behavior.

ELEMENT	DESCRIPTION
Protocol name(s)	IPX, SPX
Description	Provides transport services over the IPX networking layer: IPX for unreliable datagrams, SPX for reliable, connection-oriented message streams.
Address family	AF_IPX
Header file	Wsipx.h

This section discusses how to use Winsock with the IPX family of protocols, enabling traditional IPX applications to be ported to Winsock. IPX networks operate in a fundamentally different manner than IP networks, so such differences must be considered when using IPX/SPX.

# AF\_IPX Address Family

3/5/2021 • 2 minutes to read • [Edit Online](#)

The IPX address family defines the addressing structure for protocols that use standard IPX socket addressing. For these transports, an endpoint address consists of a network number, node address, and socket number.

The network number is an administrative domain and typically names a single Ethernet or token ring segment. The node number is a station's physical address. The combination of net and node form a unique station address that is presumed to be unique in the world. Net and node numbers are represented in ASCII text in either block or dashed notation as: '0101a040,00001b498765' or '01-01-a0-40,00-00-1b-49-87-65'. Leading zeros need not be present.

The IPX socket number is a network/transport service number much like a TCP port number and is not to be confused with the Winsock socket descriptor. IPX socket numbers are global to the end station and cannot be bound to specific net/node addresses. For instance, if the end station has two network interface cards, a bound socket can send and receive on both cards. In particular, datagram sockets would receive broadcast datagrams on both cards.

**Caution**

SOCKADDR\_IPX is 14 bytes long and is shorter than the 16-byte [sockaddr](#) reference structure. IPX/SPX implementations may accept the 16-byte length as well as the true length. If you use SOCKADDR\_IPX and a hard-coded length of 16 bytes, the implementation may assume that it has access to the 2 bytes following your structure.

FIELD	VALUE
<b>sa_family</b>	Address family AF_IPX in host order.
<b>Sa_netnum</b>	IPX network identifier in network order.
<b>Sa_nodenum</b>	Station node address, flushed right.
<b>Sa_socket</b>	IPX socket number in network order.

# IPX Family of Protocol Identifiers

3/5/2021 • 2 minutes to read • [Edit Online](#)

The `protocol` parameter in [socket](#) and [WSASocket](#) is an identifier that establishes a network type and a method for identifying the various transport protocols that run on the network. Unlike IP, IPX does not use a single protocol value for selecting a transport stack. Since there is no network requirement to use a specific value for each transport protocol, we are free to assign them in a manner convenient for Winsock applications. We avoid values 0–255 in order to avoid collisions with the corresponding PF\_INET protocol values.

NAME	VALUE	SOCKET TYPES	DESCRIPTION
Reserved	0–255		Reserved for PF_INET protocol values.
NSPROTO_IPX	1000 – 1255	SOCK_DGRAM SOCK_RAW	Datagram service for IPX.
NSPROTO_SPX	1256	SOCK_STREAM SOCK_SEQPKT	Reliable packet exchange using fixed-sized packets.

## NOTE

When NSPROTO\_SPX is specified, the SPX II protocol is automatically utilized if both endpoints are capable of doing so.

# Broadcast to Local Network

3/5/2021 • 2 minutes to read • [Edit Online](#)

A broadcast can be made to the locally attached network by setting **sa\_netnum** to binary zeros (0) and **sa\_nodenum** to binary ones (1). This broadcast may be sent to the primary network for the device, or to all locally attached networks at the option of the service provider. Broadcasts to the local network are not propagated by routers.

# All Routes Broadcast

3/5/2021 • 2 minutes to read • [Edit Online](#)

A general broadcast through the Internet is achieved by setting the **sa\_netnum** and **sa\_nodenum** fields to binary ones (1). The service provider translates this request to a type-20 packet, which IPX routers recognize and forward. The packet visits all subnets and may be duplicated several times. Receivers must handle several duplicate copies of the datagram.

Use of this broadcast type is unpopular among network administrators, so its use should be extremely limited. Many routers disable this type of broadcast, leaving parts of the subnet invisible to the packet.

# Directed Broadcast

3/5/2021 • 2 minutes to read • [Edit Online](#)

Generally considered more network friendly than an all-routes broadcast, a directed broadcast limits itself to the local network that you specify. Fill **sa\_netnum** with the target network number and **sa\_nodenumber** with binary ones. Routers forward this request to the destination network where it becomes a local broadcast. Intermediate networks do not see this packet as a broadcast.

# About Media Packet Size

3/5/2021 • 2 minutes to read • [Edit Online](#)

Media packet size affects the ability of IPX protocols to transfer data across networks and can prove challenging to deal with in a transport-independent manner. IPX does not segment packets, nor does it report when packets are dropped due to size violations. This means that some entity on the end station must maintain knowledge of the maximum packet size to be used on any given internetwork path. Traditionally, IPX datagrams and SPX connection-oriented services have left this burden to the application, while SPXII has used large Internet packet negotiation to handle it transparently.

Winsock attempts to set rational packet size limits for its various IPX protocols as described in the next section. These limits can be viewed and modified by applications through get/set socket options. When determining maximum packet size, the three areas of concern are:

- # Media packet size
- # Routable packet size
- # End-station packet size

*Media packet size* reflects the maximum packet size acceptable on any media the packet traverses to its destination. Packet size varies among differing media such as Ethernet and token ring. The amount of data space within a packet can also vary within a given media, depending on the packet header arrangement. For instance, the effective data size of an Ethernet packet varies depending on whether it is of the type Ethernet II, Ethernet 802.2, or Ethernet SNAP.

*Routable packet size* reflects the maximum packet size an intermediate router is willing to forward. Most IPX routers are built to route any size datagram as long as it remains within the media size of the sending and receiving network. However, older routers may limit maximum packet size to 576 bytes, including protocol headers.

*End-station packet size* reflects the size of the listen buffers that end stations have posted to receive incoming packets. Even when the media and router limits allow a packet through, it may be discarded by the end station if the receiving application has posted a smaller buffer. Many traditional IPX/SPX applications limit receive buffer size such that the data portion must be no larger than 512 or 1024 bytes.

# How Packet Size Affects Protocols

3/5/2021 • 2 minutes to read • [Edit Online](#)

Media packet size issues discussed in the topic, [About Media Packet Size](#), affect the various PF\_IPX protocols differently. A common strategy for handling various router and media size constraints is to use the full media size when the remote station's network number matches the local station, and fall back to the minimum packet size otherwise.

## Parameters

### NSPROTO\_IPX

Provides a datagram service; each datagram must reside within the maximum packet size. Winsock sets the maximum datagram packet size to the local media packet size minus the IPX header. Keep in mind, however, that if the packet is routed, it may hit router restrictions en route. Make sure your application can fall back to 546-byte datagrams.

### NSPROTO\_SPX

Provides stream and sequenced-packet services. Winsock IPX/SPX lets data streams and messages span multiple packets, so packet size does not limit the amount of data handled by `send` and `recv`. However, the underlying media size must be set correctly or the first large packet will be undeliverable and the connection will reset. If the target station is on the local network, Winsock sets its packet size to the media packet size. Otherwise, it defaults to 512 bytes. This size can be changed immediately after `connect` or `accept` through `setsockopt`.

### NSPROTO\_SPXII

SPXII features large Internet packet negotiation to maintain a best-fit size for packets and does not require programmer intervention. However, if the remote station does not support SPXII and negotiates down to standard SPX, the NSPROTO\_SPX rules are in effect.

PROTOCOL	MEDIA	TYPE	DATA PACKET SIZE
NSPROTO_IPX	Ethernet	Ethernet II	
		802.3	
		802.2	1466
		SNAP	
	Token Ring	4 megabit	
		16 megabit	
NSPROTO_SPX	Ethernet	Ethernet II	
		802.3	
		802.2	1454

PROTOCOL	MEDIA	TYPE	DATA PACKET SIZE
		SNAP	
	Token Ring	4 megabit	
		16 megabit	

# Winsock IPX/SPX Structures

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following list provides concise descriptions of each Winsock IPX/SPX structure. For additional information on any structure, click the structure name.

IPX/SPX STRUCTURE	DESCRIPTION
<a href="#">IPX_ADDRESS_DATA</a>	Provides information about a specific adapter to which IPX is bound.
<a href="#">IPX_NETNUM_DATA</a>	Provides information about a specified IPX network number.
<a href="#">IPX_SPXCONNSTATUS_DATA</a>	Provides information about a connected SPX socket.

# Winsock TCP/IP Annex

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section describes Transmission Control Protocol/Internet Protocol (TCP/IP) functions, data structures, and controls.

ELEMENT	DESCRIPTION
Protocol name(s)	TCP, UDP
Description	Provides transport services over the IP networking layer: UDP for unreliable datagrams, TCP for reliable, connection-oriented byte streams.
Address family	<b>AF_INET, AF_INET6</b>
Header file	<i>Ws2tcpip.h</i>

Two basic types of transport services are offered: unreliable datagrams (UDP), and reliable connection oriented-byte streams (TCP). In addition, a raw socket is optionally supported. Raw sockets allow an application to communicate through protocols other than TCP and UDP such as ICMP. Raw sockets are supported on the **AF\_INET** and **AF\_INET6** address families with some limitations.

This section covers extensions to Winsock that are specific to TCP/IP protocols. It also describes aspects of base Winsock functions that require special consideration or that may exhibit unique behavior when using TCP/IP.

# Using UNIX ioctl in Winsock

3/5/2021 • 2 minutes to read • [Edit Online](#)

The SIOCGIFCONF command provided by most UNIX implementations is supported in the form of [WSAIoctl](#) and [WSPIoctl](#) functions with the command **SIO\_GET\_INTERFACE\_LIST**. This command returns the list of configured interfaces and their parameters.

**NOTE**

Support of this command is mandatory for Windows Sockets 2-compliant TCP/IP service providers.

The parameter *lpvOutBuffer* points to the buffer in which [WSAIoctl](#) and [WSPIoctl](#) store the information about interfaces. The number of interfaces (number of structures returned in *lpvOutBuffer*) can be determined based on the actual length of the output buffer returned in *lpcbBytesReturned*.

# Winsock TCP/IP Function Details

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section presents general information about TCP/IP function details for multicast, raw sockets, and IPv6 support.

- [Multicast](#)
- [TCP/IP Raw Sockets](#)
- [IPv6 Support](#)
- [Text Representation of IPv6 Addresses](#)

# Multicast

3/5/2021 • 2 minutes to read • [Edit Online](#)

Generic Winsock multipoint functions support IP multicast. However, the TCP/IP transport providers that support multicast must also provide Berkeley Software Distribution (BSD) style–multicast support by supporting the corresponding multicast options. This simplifies the porting of existing multicast applications to Windows Sockets 2.

# TCP/IP Raw Sockets

3/5/2021 • 9 minutes to read • [Edit Online](#)

A raw socket is a type of socket that allows access to the underlying transport provider. This topic focuses only on raw sockets and the IPv4 and IPv6 protocols. This is because most other protocols with the exception of ATM do not support raw sockets. To use raw sockets, an application needs to have detailed information on the underlying protocol being used.

Winsock service providers for the IP protocol may support a socket *type* of **SOCK\_RAW**. The Windows Sockets 2 provider for TCP/IP included on Windows supports this **SOCK\_RAW** socket type.

There are two basic types of such raw sockets:

- The first type uses a known protocol type written in the IP header that is recognized by a Winsock service provider. An example of the first type of socket is a socket for the ICMP protocol (IP protocol type = 1) or the ICMPv6 protocol (IP protocol type = 58).
- The second type allows any protocol type to be specified. An example of the second type would be an experimental protocol that is not directly supported by the Winsock service provider such as the Stream Control Transmission Protocol (SCTP).

## Determining if Raw Sockets are Supported

If a Winsock service provider supports **SOCK\_RAW** sockets for the AF\_INET or AF\_INET6 address families, the socket type of **SOCK\_RAW** should be included in the **WSAPROTOCOL\_INFO** structure returned by **WSAEnumProtocols** function for one or more of the available transport providers.

The **iAddressFamily** member in the **WSAPROTOCOL\_INFO** structure should specify AF\_INET or AF\_INET6 and the **iSocketType** member of the **WSAPROTOCOL\_INFO** structure should specify **SOCK\_RAW** for one of the transport providers.

The **iProtocol** member of the **WSAPROTOCOL\_INFO** structure may be set to **IPPROTO\_IP**. The **iProtocol** member of the **WSAPROTOCOL\_INFO** structure may also be set to zero if the service provider allows an application to use a **SOCK\_RAW** socket type for other network protocols other than the Internet Protocol for the address family.

The other members in the **WSAPROTOCOL\_INFO** structure indicate other properties of the protocol support for **SOCK\_RAW** and indicate how a socket of **SOCK\_RAW** should be treated. These other members of the **WSAPROTOCOL\_INFO** for **SOCK\_RAW** normally specify that the protocol is connectionless, message-oriented, supports broadcast/multicast (the **XP1\_CONNECTIONLESS**, **XP1\_MESSAGE\_ORIENTED**, **XP1\_SUPPORT\_BROADCAST**, and **XP1\_SUPPORT\_MULTIPOINT** bits are set in the **dwServiceFlags1** member), and can have a maximum message size of 65,467 bytes.

On Windows XP and later, the **NetSh.exe** command can be used to determine if raw sockets are supported. The following command run from a CMD window will display data from the Winsock catalog on the console:

```
netsh winsock show catalog
```

The output will include a list that contains some of the data from the **WSAPROTOCOL\_INFO** structures supported on the local computer. Search for the term **RAW/IP** or **RAW/IPv6** in the **Description** field to find those protocols that support raw sockets.

## Creating a Raw Socket

To create a socket of type **SOCK\_RAW**, call the [socket](#) or [WSASocket](#) function with the *af* parameter (address family) set to AF\_INET or AF\_INET6, the *type* parameter set to **SOCK\_RAW**, and the *protocol* parameter set to the protocol number required. The *protocol* parameter becomes the protocol value in the IP header (SCTP is 132, for example).

**NOTE**

An application may not specify zero (0) as the *protocol* parameter for the [socket](#), [WSASocket](#), and [WSPSocket](#) functions if the *type* parameter is set to **SOCK\_RAW**.

Raw sockets offer the capability to manipulate the underlying transport, so they can be used for malicious purposes that pose a security threat. Therefore, only members of the Administrators group can create sockets of type **SOCK\_RAW** on Windows 2000 and later.

## Send and Receive Operations

Once an application creates a socket of type **SOCK\_RAW**, this socket may be used to send and receive data. All packets sent or received on a socket of type **SOCK\_RAW** are treated as datagrams on an unconnected socket.

The following rules apply to the operations over **SOCK\_RAW** sockets:

- The [sendto](#) or [WSASendTo](#) function is normally used to send data on a socket of type **SOCK\_RAW**. The destination address can be any valid address in the socket's address family, including a broadcast or multicast address. To send to a broadcast address, an application must have used [setsockopt](#) with SO\_BROADCAST enabled. Otherwise, [sendto](#) or [WSASendTo](#) will fail with the error code [WSAEACCES](#). For IP, an application can send to any multicast address (without becoming a group member).
- When sending IPv4 data, an application has a choice on whether to specify the IPv4 header at the front of the outgoing datagram for the packet. If the **IP\_HDRINCL** socket option is set to true for an IPv4 socket (address family of AF\_INET), the application must supply the IPv4 header in the outgoing data for send operations. If this socket option is false (the default setting), then the IPv4 header should not be included in the outgoing data for send operations.
- When sending IPv6 data, an application has a choice on whether to specify the IPv6 header at the front of the outgoing datagram for the packet. If the **IPV6\_HDRINCL** socket option is set to true for an IPv6 socket (address family of AF\_INET6), the application must supply the IPv6 header in the outgoing data for send operations. The default setting for this option is false. If this socket option is false (the default setting), then the IPv6 header should not be included in the outgoing data for send operations. For IPv6, there should be no need to include the IPv6 header. If information is available using socket functions, then the IPv6 header should not be included to avoid compatibility problems in the future. These issues are discussed in RFC 3542 published by the IETF. Using the **IPV6\_HDRINCL** socket option is not recommended and may be deprecated in future.
- The [recvfrom](#) or [WSARecvFrom](#) function is normally used to receive data on a socket of type **SOCK\_RAW**. Both of these functions have an option to return the source IP address where the packet was sent from. The received data is a datagram from an unconnected socket.
- For IPv4 (address family of AF\_INET), an application receives the IP header at the front of each received datagram regardless of the **IP\_HDRINCL** socket option.
- For IPv6 (address family of AF\_INET6), an application receives everything after the last IPv6 header in each received datagram regardless of the **IPV6\_HDRINCL** socket option. The application does not receive any IPv6 headers using a raw socket.

- Received datagrams are copied into all **SOCK\_RAW** sockets that satisfy the following conditions:
  - The protocol number specified in the *proto*/parameter when the socket was created should match the protocol number in the IP header of the received datagram.
  - If a local IP address is defined for the socket, it should correspond to the destination address as specified in the IP header of the received datagram. An application may specify the local IP address by calling the **bind** function. If no local IP address is specified for the socket, the datagrams are copied into the socket regardless of the destination IP address in the IP header of the received datagram.
  - If a foreign address is defined for the socket, it should correspond to the source address as specified in the IP header of the received datagram. An application may specify the foreign IP address by calling the **connect** or **WSAConnect** function. If no foreign IP address is specified for the socket, the datagrams are copied into the socket regardless of the source IP address in the IP header of the received datagram.

It is important to understand that some sockets of type **SOCK\_RAW** may receive many unexpected datagrams. For example, a PING program may create a socket of type **SOCK\_RAW** to send ICMP echo requests and receive responses. While the application is expecting ICMP echo responses, all other ICMP messages (such as ICMP HOST\_UNREACHABLE) may also be delivered to this application. Moreover, if several **SOCK\_RAW** sockets are open on a computer at the same time, the same datagrams may be delivered to all the open sockets. An application must have a mechanism to recognize the datagrams of interest and to ignore all others. For a PING program, such a mechanism might include inspecting the received IP header for unique identifiers in the ICMP header (the application's process ID, for example).

#### NOTE

To use a socket of type **SOCK\_RAW** requires administrative privileges. Users running Winsock applications that use raw sockets must be a member of the Administrators group on the local computer, otherwise raw socket calls will fail with an error code of **WSAEACCES**. On Windows Vista and later, access for raw sockets is enforced at socket creation. In earlier versions of Windows, access for raw sockets is enforced during other socket operations.

## Common Uses of Raw Sockets

One common use of raw sockets are troubleshooting applications that need to examine IP packets and headers in detail. For example, a raw socket can be used with the **SIO\_RCVALL** IOCTL to enable a socket to receive all IPv4 or IPv6 packets passing through a network interface. For more information, see the **SIO\_RCVALL** reference.

## Limitations on Raw Sockets

On Windows 7, Windows Vista, Windows XP with Service Pack 2 (SP2), and Windows XP with Service Pack 3 (SP3), the ability to send traffic over raw sockets has been restricted in several ways:

- TCP data cannot be sent over raw sockets.
- UDP datagrams with an invalid source address cannot be sent over raw sockets. The IP source address for any outgoing UDP datagram must exist on a network interface or the datagram is dropped. This change was made to limit the ability of malicious code to create distributed denial-of-service attacks and limits the ability to send spoofed packets (TCP/IP packets with a forged source IP address).
- A call to the **bind** function with a raw socket for the **IPPROTO\_TCP** protocol is not allowed.

**NOTE**

The **bind** function with a raw socket is allowed for other protocols (IPPROTO\_IP, IPPROTO\_UDP, or IPPROTO\_SCTP, for example).

These above restrictions do not apply to Windows Server 2008 R2, Windows Server 2008 , Windows Server 2003, or to versions of the operating system earlier than Windows XP with SP2.

**NOTE**

The Microsoft implementation of TCP/IP on Windows is capable of opening a raw UDP or TCP socket based on the above restrictions. Other Winsock providers may not support the use of raw sockets.

There are further limitations for applications that use a socket of type **SOCK\_RAW**. For example, all applications listening for a specific protocol will receive all packets received for this protocol. This may not be what is desired for multiple applications using a protocol. This is also not suitable for high-performance applications. To get around these issues, it may be required to write a Windows network protocol driver (device driver) for the specific network protocol. On Windows Vista and later, Winsock Kernel (WSK), a new transport-independent kernel mode Network Programming Interface can be used to write a network protocol driver. On Windows Server 2003 and earlier, a Transport Driver Interface (TDI) provider and a Winsock helper DLL can be written to support the network protocol. The network protocol would then be added to the Winsock catalog as a supported protocol. This allows multiple applications to open sockets for this specific protocol and the device driver can keep track of which socket receives specific packets and errors. For information on writing a network protocol provider, see the sections on WSK and TDI in the Windows Driver Kit (WDK).

Applications also need to be aware of the impact that firewall settings may have on sending and receiving packets using raw sockets.

# IPv6 Support

3/5/2021 • 2 minutes to read • [Edit Online](#)

In order to support both IPv4 and IPv6 on Windows XP with Service Pack 1 (SP1) and on Windows Server 2003, an application has to create two sockets, one socket for use with IPv4 and one socket for use with IPv6. These two sockets must be handled separately by the application.

If a TCP/IP service provider on Windows XP with SP1 and on Windows Server 2003 supports IPv4 and IPv6 addressing, it must create two separate sockets and listen separately on these sockets:

- Once for IPv4.
- Once for the IPv6 address family.

Windows Vista and later offer the ability to create a single IPv6 socket which can handle both IPv6 and IPv4 traffic. For example, a TCP listening socket for IPv6 is created, put into dual stack mode, and bound to port 5001. This dual-stack socket can accept connections from IPv6 TCP clients connecting to port 5001 and from IPv4 TCP clients connecting to port 5001. This feature allows for greatly simplified application design and reduces the resource overhead required of posting operations on two separate sockets. However, there are some restrictions that must be met in order to use a dual-stack socket. For more information, see [Dual-Stack Sockets](#).

**WSAEnumProtocols** returns two **WSAPROTOCOL\_INFO** structures for each of the supported socket types (SOCK\_STREAM, SOCK\_DGRAM, SOCK\_RAW). The **iAddressFamily** must be set to AF\_INET for IPv4 addressing, and to AF\_INET6 for IPv6 addressing.

The IPv6 addresses are described in the following structures.

```
struct in_addr6 {
    u_char     s6_addr[16];           /* IPv6 address */
};

struct sockaddr_in6 {
    short      sin6_family;        /* AF_INET6 */
    u_short    sin6_port;          /* Transport level port number */
    u_long     sin6_flowinfo;      /* IPv6 flow information */
    struct in_addr6  sin6_addr;    /* IPv6 address */
    u_long     sin6_scope_id;      /* set of interfaces for a scope */
};
```

If an application uses Windows Sockets 1.1 functions and wants to use IPv6 addresses, it may continue to use all the old functions that take the **sockaddr** structure as one of the parameters (**bind**, **connect**, **sendto**, and **recvfrom**, **accept**, and so forth). The only change that is required is to use **sockaddr\_in6** instead of **sockaddr\_in**.

However, the name resolution functions (**gethostbyname**, **gethostbyaddr**, and so forth) and address conversion functions (**inet\_addr**, **inet\_ntoa**) cannot be reused because they assume an IP address is 4 bytes in length. An application that wants to perform name resolution and address conversion for IPv6 addresses must use Windows Sockets 2-specific functions. Many new functions have been introduced to enable Windows Sockets 2 applications to take advantage of IPv6, including the **getaddrinfo** and **getnameinfo** functions.

For more information on how to enable IPv6 capabilities in an application, see the [IPv6 Guide for Windows Sockets Applications](#).



# Text Representation of IPv6 Addresses

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section is copied from "IPv6 Addressing Architecture" by R. Hinden and S. Deering. There are three conventional forms for representing IPv6 addresses as text strings:

- The preferred form is `x:x:x:x:x:x`, where the 'x's are the hexadecimal values of the eight 16-bit pieces of the address.

Examples:

`FEDC:BA98:7654:3210:FEDC:BA98:7654:3210 1080:0:0:8:800:200C:417A`

## NOTE

It is not necessary to write the leading zeros in an individual field, but there must be at least one numeral in every field (except for the case described in the second form).

- Due to the method of allocating certain styles of IPv6 addresses, it is common for addresses to contain long strings of zero bits. In order to make writing addresses containing zero bits easier, a special syntax is available to compress the zeros. The use of double colons ("::") indicates multiple groups of 16-bits of zeros.

For example, the multicast address

`FF01:0:0:0:0:0:43`

may be represented as:

`FF01::43`

The double quotation marks ("::") can only appear once in an address. They can be used to compress leading or trailing zeros in an address.

- An alternative form that may be more convenient when dealing with a mixed environment of IPv4 and IPv6 nodes is `x:x:x:x:d.d.d.d`, where the 'x's are the hexadecimal values of the six high-order 16-bit pieces of the address, and the 'd's are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation).

Examples:

`0:0:0:0:0:13.1.68.3 0:0:0:0:FFFF:129.144.52.38`

or in compressed form:

`::13.1.68.3 ::FFFF:129.144.52.38`

# Winsock TCP/IP Structures

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following list provides concise descriptions of each Winsock TCP/IP structure. For additional information on any structure, click the structure name.

TCP/IP STRUCTURE	DESCRIPTION
<a href="#">INTERFACE_INFO</a>	Used with the ioctl command to obtain information about an interface IP address.
<a href="#">INTERFACE_INFO_EX</a>	Used with the <b>SIO_GET_INTERFACE_LIST</b> IOCTL command to obtain information about an interface IP address, including IPv6 addresses.
<a href="#">sockaddr_gen</a>	Provides generic socket address information. Used with the <a href="#">INTERFACE_INFO</a> structure.

# Winsock Enumerations

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following list provides concise descriptions of each Winsock enumeration. For additional information on any enumeration, click the enumeration name.

ENUMERATION	DESCRIPTION
<a href="#">CONTROL_CHANNEL_TRIGGER_STATUS</a>	Specifies the status from a query for the <a href="#">REAL_TIME_NOTIFICATION_CAPABILITY</a> transport setting for a TCP socket that is used with <a href="#">ControlChannelTrigger</a> to receive background network notifications in a Windows Runtime application.
<a href="#">eWINDOW_ADVANCE_METHOD</a>	Specifies the window advance mode used for Reliable Multicast.
<a href="#">GUARANTEE</a>	No longer used.
<a href="#">MULTICAST_MODE_TYPE</a>	Specifies the filter mode for multicast group addresses.
<a href="#">NAPI_PROVIDER_LEVEL</a>	Specifies the provider authority level of a NS_EMAIL namespace provider for a given domain.
<a href="#">NAPI_PROVIDER_TYPE</a>	Specifies the type of hosting expected for a namespace provider.
<a href="#">RIO_NOTIFICATION_COMPLETION_TYPE</a>	Specifies the type of completion queue notifications to use with the <a href="#">RIONotify</a> function when sending or receiving data using the Winsock registered I/O extensions.
<a href="#">SOCKET_SECURITY_PROTOCOL</a>	Indicates the type of security protocol to be used on a socket.
<a href="#">SOCKET_USAGE_TYPE</a>	Used to specified the usage type for the socket.
<a href="#">TCPSTATE</a>	Indicates the possible states of a Transmission Control Protocol (TCP) connection.
<a href="#">WSAECOMPARATOR</a>	Used for version-comparison semantics.
<a href="#">WSC_PROVIDER_INFO_TYPE</a>	Specifies the information class of a layered service protocol (LSP).

# GUARANTEE

3/5/2021 • 2 minutes to read • [Edit Online](#)

The Windows Sockets **GUARANTEE** enumeration is no longer used. For information regarding the Windows implementation of Quality of Service, and the associated API, refer to [Quality of Service \(QoS\)](#) in the Platform Software Development Kit (SDK).

# Winsock functions

3/5/2021 • 11 minutes to read • [Edit Online](#)

The following list provides concise descriptions of each Winsock function. For additional information on any function, click the function name.

FUNCTION	DESCRIPTION
<a href="#">accept</a>	Permits an incoming connection attempt on a socket.
<a href="#">AcceptEx</a>	Accepts a new connection, returns the local and remote address, and receives the first block of data sent by the client application.
<a href="#">bind</a>	Associates a local address with a socket.
<a href="#">closesocket</a>	Closes an existing socket.
<a href="#">connect</a>	Establishes a connection to a specified socket.
<a href="#">ConnectEx</a>	Establishes a connection to a specified socket, and optionally sends data once the connection is established. Only supported on connection-oriented sockets.
<a href="#">DisconnectEx</a>	Closes a connection on a socket, and allows the socket handle to be reused.
<a href="#">EnumProtocols</a>	Retrieves information about a specified set of network protocols that are active on a local host.
<a href="#">freeaddrinfo</a>	Frees address information that the <a href="#">getaddrinfo</a> function dynamically allocates in <a href="#">addrinfo</a> structures.
<a href="#">FreeAddrInfoEx</a>	Frees address information that the <a href="#">GetAddrInfoEx</a> function dynamically allocates in <a href="#">addrinfoex</a> structures.
<a href="#">FreeAddrInfoW</a>	Frees address information that the <a href="#">GetAddrInfoW</a> function dynamically allocates in <a href="#">addrinfoW</a> structures.
<a href="#">gai_strerror</a>	Assists in printing error messages based on the EAI_* errors returned by the <a href="#">getaddrinfo</a> function.
<a href="#">GetAcceptExSockaddrs</a>	Parses the data obtained from a call to the <a href="#">AcceptEx</a> function.
<a href="#">GetAddressByName</a>	Queries a namespace, or a set of default namespaces, to retrieve 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 <a href="#">bind</a> function.

FUNCTION	DESCRIPTION
<a href="#">getaddrinfo</a>	Provides protocol-independent translation from an ANSI host name to an address.
<a href="#">GetAddrInfoEx</a>	Provides protocol-independent name resolution with additional parameters to qualify which name space providers should handle the request.
<a href="#">GetAddrInfoExCancel</a>	Cancels an asynchronous operation by the <a href="#">GetAddrInfoEx</a> function.
<a href="#">GetAddrInfoExOverlappedResult</a>	Gets the return code for an <b>OVERLAPPED</b> structure used by an asynchronous operation for the <a href="#">GetAddrInfoEx</a> function.
<a href="#">GetAddrInfoW</a>	Provides protocol-independent translation from a Unicode host name to an address.
<a href="#">gethostbyaddr</a>	Retrieves the host information corresponding to a network address.
<a href="#">gethostbyname</a>	Retrieves host information corresponding to a host name from a host database. Deprecated: use <a href="#">getaddrinfo</a> instead.
<a href="#">gethostname</a>	Retrieves the standard host name for the local computer.
<a href="#">GetHostNameW</a>	Retrieves the standard host name for the local computer as a Unicode string.
<a href="#">getipv4sourcefilter</a>	Retrieves the multicast filter state for an IPv4 socket.
<a href="#">GetNameByType</a>	Retrieves the name of a network service for the specified service type.
<a href="#">getnameinfo</a>	Provides name resolution from an IPv4 or IPv6 address to an ANSI host name and from a port number to the ANSI service name.
<a href="#">GetNameInfoW</a>	Provides name resolution from an IPv4 or IPv6 address to a Unicode host name and from a port number to the Unicode service name.
<a href="#">getpeername</a>	Retrieves the address of the peer to which a socket is connected.
<a href="#">getprotobynumber</a>	Retrieves the protocol information corresponding to a protocol name.
<a href="#">getprotobyname</a>	Retrieves protocol information corresponding to a protocol number.
<a href="#">getservbyname</a>	Retrieves service information corresponding to a service name and protocol.

FUNCTION	DESCRIPTION
<a href="#">getservbyport</a>	Retrieves service information corresponding to a port and protocol.
<a href="#">GetService</a>	Retrieves information about a network service in the context of a set of default namespaces or a specified namespace.
<a href="#">getsockname</a>	Retrieves the local name for a socket.
<a href="#">getsockopt</a>	Retrieves a socket option.
<a href="#">getsourcefilter</a>	Retrieves the multicast filter state for an IPv4 or IPv6 socket.
<a href="#">GetTypeByName</a>	Retrieves a service type GUID for a network service specified by name.
<a href="#">htond</a>	Converts a <b>double</b> from host to TCP/IP network byte order (which is big-endian).
<a href="#">htonf</a>	Converts a <b>float</b> from host to TCP/IP network byte order (which is big-endian).
<a href="#">htonl</a>	Converts a <b>u_long</b> from host to TCP/IP network byte order (which is big-endian).
<a href="#">htonll</a>	Converts an <b>unsigned __int64</b> from host to TCP/IP network byte order (which is big-endian).
<a href="#">htons</a>	Converts a <b>u_short</b> from host to TCP/IP network byte order (which is big-endian).
<a href="#">inet_addr</a>	Converts a string containing an (Ipv4) Internet Protocol dotted address into a proper address for the <b>in_addr</b> structure.
<a href="#">inet_ntoa</a>	Converts an (IPv4) Internet network address into a string in Internet standard dotted format.
<a href="#">InetNtop</a>	converts an IPv4 or IPv6 Internet network address into a string in Internet standard format. The ANSI version of this function is <a href="#">inet_ntop</a> .
<a href="#">InetPton</a>	Converts an IPv4 or IPv6 Internet network address in its standard text presentation form into its numeric binary form. The ANSI version of this function is <a href="#">inet_pton</a> .
<a href="#">ioctlsocket</a>	Controls the I/O mode of a socket.
<a href="#">listen</a>	Places a socket a state where it is listening for an incoming connection.
<a href="#">ntohd</a>	Converts an <b>unsigned __int64</b> from TCP/IP network order to host byte order (which is little-endian on Intel processors) and returns a <b>double</b> .

FUNCTION	DESCRIPTION
<a href="#">ntohf</a>	Converts an <b>unsigned __int32</b> from TCP/IP network order to host byte order (which is little-endian on Intel processors) and returns a <b>float</b> .
<a href="#">ntohl</a>	Converts a <b>u_long</b> from TCP/IP network order to host byte order (which is little-endian on Intel processors).
<a href="#">ntohll</a>	Converts an <b>unsigned __int64</b> from TCP/IP network order to host byte order (which is little-endian on Intel processors).
<a href="#"> ntohs</a>	Converts a <b>u_short</b> from TCP/IP network byte order to host byte order (which is little-endian on Intel processors).
<a href="#">recv</a>	Receives data from a connected or bound socket.
<a href="#">recvfrom</a>	Receives a datagram and stores the source address.
<a href="#">RIOCloseCompletionQueue</a>	Closes an existing completion queue used for I/O completion notification by send and receive requests with the Winsock registered I/O extensions.
<a href="#">RIOCreateCompletionQueue</a>	Creates an I/O completion queue of a specific size for use with the Winsock registered I/O extensions.
<a href="#">RIOCreateRequestQueue</a>	Creates a registered I/O socket descriptor using a specified socket and I/O completion queues for use with the Winsock registered I/O extensions.
<a href="#">RIODequeueCompletion</a>	Removes entries from an I/O completion queue for use with the Winsock registered I/O extensions.
<a href="#">RIODeregisterBuffer</a>	Deregisters a registered buffer used with the Winsock registered I/O extensions.
<a href="#">RIONotify</a>	Registers the method to use for notification behavior with an I/O completion queue for use with the Winsock registered I/O extensions.
<a href="#">RIOReceive</a>	Receives network data on a connected registered I/O TCP socket or a bound registered I/O UDP socket for use with the Winsock registered I/O extensions.
<a href="#">RIOReceiveEx</a>	Receives network data on a connected registered I/O TCP socket or a bound registered I/O UDP socket with additional options for use with the Winsock registered I/O extensions.
<a href="#">RIORegisterBuffer</a>	Registers a <b>RIO_BUFFERID</b> , a registered buffer descriptor, with a specified buffer for use with the Winsock registered I/O extensions.
<a href="#">RIOResizeCompletionQueue</a>	Resizes an I/O completion queue to be either larger or smaller for use with the Winsock registered I/O extensions.

FUNCTION	DESCRIPTION
<a href="#">RIOResizeRequestQueue</a>	Resizes a request queue to be either larger or smaller for use with the Winsock registered I/O extensions.
<a href="#">RIOSend</a>	Sends network data on a connected registered I/O TCP socket or a bound registered I/O UDP socket for use with the Winsock registered I/O extensions.
<a href="#">RIOSendEx</a>	Sends network data on a connected registered I/O TCP socket or a bound registered I/O UDP socket with additional options for use with the Winsock registered I/O extensions.
<a href="#">select</a>	Determines the status of one or more sockets, waiting if necessary, to perform synchronous I/O.
<a href="#">send</a>	Sends data on a connected socket.
<a href="#">sendto</a>	Sends data to a specific destination.
<a href="#">SetAddrInfoEx</a>	Registers a host and service name along with associated addresses with a specific namespace provider.
<a href="#">setipv4sourcefilter</a>	Sets the multicast filter state for an IPv4 socket.
<a href="#">SetService</a>	Registers or removes from the registry a network service within one or more namespaces. Can also add or remove a network service type within one or more namespaces.
<a href="#">SetSocketMediaStreamingMode</a>	Indicates whether the network is to be used for transferring streaming media that requires quality of service.
<a href="#">setsockopt</a>	Sets a socket option.
<a href="#">setsourcefilter</a>	Sets the multicast filter state for an IPv4 or IPv6 socket.
<a href="#">shutdown</a>	Disables sends or receives on a socket.
<a href="#">socket</a>	Creates a socket that is bound to a specific service provider.
<a href="#">TransmitFile</a>	Transmits file data over a connected socket handle.
<a href="#">TransmitPackets</a>	Transmits in-memory data or file data over a connected socket.
<a href="#">WSAAccept</a>	Conditionally accepts a connection based on the return value of a condition function, provides quality of service flow specifications, and allows the transfer of connection data.
<a href="#">WSAAddressToString</a>	Converts all components of a <a href="#">sockaddr</a> structure into a human-readable string representation of the address.
<a href="#">WSAAyncGetHostByAddr</a>	Asynchronously retrieves host information that corresponds to an address.

FUNCTION	DESCRIPTION
<a href="#">WSAAAsyncGetHostByName</a>	Asynchronously retrieves host information that corresponds to a host name.
<a href="#">WSAAAsyncGetProtoByName</a>	Asynchronously retrieves protocol information that corresponds to a protocol name.
<a href="#">WSAAAsyncGetProtoByNumber</a>	Asynchronously retrieves protocol information that corresponds to a protocol number.
<a href="#">WSAAAsyncGetServByName</a>	Asynchronously retrieves service information that corresponds to a service name and port.
<a href="#">WSAAAsyncGetServByPort</a>	Asynchronously retrieves service information that corresponds to a port and protocol.
<a href="#">WSAAsyncSelect</a>	Requests Windows message-based notification of network events for a socket.
<a href="#">WSACancelAsyncRequest</a>	Cancels an incomplete asynchronous operation.
<a href="#">WSACleanup</a>	Terminates use of the Ws2_32.DLL.
<a href="#">WSACloseEvent</a>	Closes an open event object handle.
<a href="#">WSAConnect</a>	Establishes a connection to another socket application, exchanges connect data, and specifies needed quality of service based on the specified <b>FLOWSPEC</b> structure.
<a href="#">WSAConnectByList</a>	Establishes a connection to one out of a collection of possible endpoints represented by a set of destination addresses (host names and ports).
<a href="#">WSAConnectByName</a>	Establishes a connection to another socket application on a specified host and port
<a href="#">WSACreateEvent</a>	Creates a new event object.
<a href="#">WSADeleteSocketPeerTargetName</a>	Removes the association between a peer target name and an IP address for a socket.
<a href="#">WSADuplicateSocket</a>	Returns a structure that can be used to create a new socket descriptor for a shared socket.
<a href="#">WSAEnumNameSpaceProviders</a>	Retrieves information about available namespaces.
<a href="#">WSAEnumNameSpaceProvidersEx</a>	Retrieves information about available namespaces.
<a href="#">WSAEnumNetworkEvents</a>	Discovers occurrences of network events for the indicated socket, clear internal network event records, and reset event objects (optional).
<a href="#">WSAEnumProtocols</a>	Retrieves information about available transport protocols.

FUNCTION	DESCRIPTION
<a href="#">WSAEventSelect</a>	Specifies an event object to be associated with the specified set of FD_XXX network events.
<a href="#">__WSAFDIsSet</a>	Specifies whether a socket is included in a set of socket descriptors.
<a href="#">WSAGetFailConnectOnIcmpError</a>	Queries the state of the <a href="#">TCP_FAIL_CONNECT_ON_ICMP_ERROR</a> socket option.
<a href="#">WSAGetIcmpErrorInfo</a>	Queries the source address of an ICMP error received on a TCP socket during connection setup.
<a href="#">WSAGetIPUserMtu</a>	Retrieves the user-defined IP layer MTU for a socket.
<a href="#">WSAGetLastError</a>	Returns the error status for the last operation that failed.
<a href="#">WSAGetOverlappedResult</a>	Retrieves the results of an overlapped operation on the specified socket.
<a href="#">WSAGetQOSByName</a>	Initializes a <a href="#">QOS</a> structure based on a named template, or it supplies a buffer to retrieve an enumeration of the available template names.
<a href="#">WSAGetServiceClassInfo</a>	Retrieves the class information (schema) pertaining to a specified service class from a specified namespace provider.
<a href="#">WSAGetServiceClassNameByClassId</a>	Retrieves the name of the service associated with the specified type.
<a href="#">WSAGetUdpRecvMaxCoalescedSize</a>	Retrieves the maximum size of a received, coalesced message for a UDP socket.
<a href="#">WSAGetUdpSendMessageSize</a>	Retrieves the segmentation message size for a UDP socket.
<a href="#">WSAhtonl</a>	Converts a u_long from host byte order to network byte order.
<a href="#">WSAhtons</a>	Converts a u_short from host byte order to network byte order.
<a href="#">WSAImpersonateSocketPeer</a>	Used to impersonate the security principal corresponding to a socket peer in order to perform application-level authorization.
<a href="#">WSAInstallServiceClass</a>	Registers a service class schema within a namespace.
<a href="#">WSAioctl</a>	Controls the mode of a socket.
<a href="#">WSAJoinLeaf</a>	Joins a leaf node into a multipoint session, exchanges connect data, and specifies needed quality of service based on the specified structures.

FUNCTION	DESCRIPTION
<a href="#">WSALookupServiceBegin</a>	Initiates a client query that is constrained by the information contained within a <a href="#">WSAQUERYSET</a> structure.
<a href="#">WSALookupServiceEnd</a>	Frees the handle used by previous calls to <a href="#">WSALookupServiceBegin</a> and <a href="#">WSALookupServiceNext</a> .
<a href="#">WSALookupServiceNext</a>	Retrieve the requested service information.
<a href="#">WSANSPIoctl</a>	Developers to make I/O control calls to a registered namespace.
<a href="#">WSANtohl</a>	Converts a u_long from network byte order to host byte order.
<a href="#">WSANtohs</a>	Converts a u_short from network byte order to host byte order.
<a href="#">WSAPoll</a>	Determines status of one or more sockets.
<a href="#">WSAProviderConfigChange</a>	Notifies the application when the provider configuration is changed.
<a href="#">WSAQuerySocketSecurity</a>	Queries information about the security applied to a connection on a socket.
<a href="#">WSARecv</a>	Receives data from a connected socket.
<a href="#">WSARecvDisconnect</a>	Terminates reception on a socket, and retrieves the disconnect data if the socket is connection oriented.
<a href="#">WSARecvEx</a>	Receives data from a connected socket.
<a href="#">WSARecvFrom</a>	Receives a datagram and stores the source address.
<a href="#">LPFN_WSARECVMSG (WSARecvMsg)</a>	Receives data and optional control information from connected and unconnected sockets.
<a href="#">WSARemoveServiceClass</a>	Permanently removes the service class schema from the registry.
<a href="#">WSAResetEvent</a>	Resets the state of the specified event object to nonsignaled.
<a href="#">WSARevertImpersonation</a>	Terminates the impersonation of a socket peer.
<a href="#">WSASend</a>	Sends data on a connected socket.
<a href="#">WSASendDisconnect</a>	Initiates termination of the connection for the socket and sends disconnect data.
<a href="#">WSASendMsg</a>	Sends data and optional control information from connected and unconnected sockets.

FUNCTION	DESCRIPTION
<a href="#">WSASendTo</a>	Sends data to a specific destination, using overlapped I/O where applicable.
<a href="#">WSASetEvent</a>	Sets the state of the specified event object to signaled.
<a href="#">WSASetFailConnectOnIcmpError</a>	Sets the state of the <a href="#">TCP_FAIL_CONNECT_ON_ICMP_ERROR</a> socket option.
<a href="#">WSASetIPUserMtu</a>	Sets the user-defined IP layer MTU on a socket.
<a href="#">WSASetLastError</a>	Sets the error code.
<a href="#">WSASetService</a>	Registers or removes from the registry a service instance within one or more namespaces.
<a href="#">WSASetSocketPeerTargetName</a>	Used to specify the peer target name (SPN) that corresponds to a peer IP address. This target name is meant to be specified by client applications to securely identify the peer that should be authenticated.
<a href="#">WSASetSocketSecurity</a>	Enables and applies security for a socket.
<a href="#">WSASetUdpRecvMaxCoalescedSize</a>	Sets the maximum size of a coalesced message set on a UDP socket.
<a href="#">WSASetUdpSendMessageSize</a>	Sets the segmentation message size on a UDP socket.
<a href="#">WSASocket</a>	Creates a socket that is bound to a specific transport-service provider.
<a href="#">WSAStartup</a>	Initiates use of WS2_32.DLL by a process.
<a href="#">WSAStringToAddress</a>	Converts a numeric string to a <a href="#">sockaddr</a> structure.
<a href="#">WSAWaitForMultipleEvents</a>	Returns either when one or all of the specified event objects are in the signaled state, or when the time-out interval expires.

# Winsock Structures

3/5/2021 • 5 minutes to read • [Edit Online](#)

The following list provides concise descriptions of each Winsock structure and typedef. For additional information on any structure or typedef, click the structure or typedef name.

STRUCTURE	DESCRIPTION
<a href="#">addrinfo</a>	Used by the <a href="#">getaddrinfo</a> function to hold host address information.
<a href="#">addrinfoW</a>	Used by the <a href="#">GetAddrInfoW</a> function to hold host address information.
<a href="#">addrinfoex</a>	Used by the <a href="#">GetAddrInfoEx</a> function to hold host address information.
<a href="#">addrinfoex2</a>	Used by the <a href="#">GetAddrInfoEx</a> function to hold host address information when both a canonical name and a fully qualified domain name have been requested.
<a href="#">addrinfoex3</a>	Used by the <a href="#">GetAddrInfoEx</a> function to hold host address information when a specific network interface has been requested.
<a href="#">addrinfoex4</a>	Used by the <a href="#">GetAddrInfoEx</a> function to hold host address information when a handle to the fully qualified domain name has been requested.
<a href="#">AFPROTOCOLS</a>	Supplies a list of protocols to which application programmers can constrain queries. Used for query purposes only.
<a href="#">BLOB</a>	Contains information about a block of data. Derived from Binary Large Object.
<a href="#">CSADDR_INFO</a>	Contains Winsock address information for a network service or namespace provider.
<a href="#">fd_set</a>	Used by various Winsock functions and service providers, such as <a href="#">select</a> , to place sockets into a "set" for various purposes, such as testing a given socket for readability.
<a href="#">GROUP_FILTER</a>	Provides multicast filtering parameters for multicast IPv6 or IPv4 addresses.
<a href="#">GROUP_REQ</a>	Provides multicast group information for IPv6 or IPv4 addresses.
<a href="#">GROUP_SOURCE_REQ</a>	Provides multicast group information for IPv6 or IPv4 addresses that includes the source IP address.

STRUCTURE	DESCRIPTION
<a href="#">hostent</a>	Stores information about a given host, such as host name, IP address, and so forth.
<a href="#">in_addr</a>	Represents an IPv4 Internet address.
<a href="#">in_pktinfo</a>	Stores received packet address information, and is used by Windows to return information about received packets.
<a href="#">in6_addr</a>	Represents an IPv6 Internet address.
<a href="#">in6_pktinfo</a>	Stores received IPv6 packet address information, and is used by Windows to return information about received packets.
<a href="#">INET_PORT_RANGE</a>	Provides input data used by the <b>SIO_ACQUIRE_PORT_RESERVATION</b> ioctl to acquire a runtime reservation for a block of TCP or UDP ports.
<a href="#">INET_PORT_RESERVATION_INSTANCE</a>	Contains a port reservation and a token for a block of TCP or UDP ports.
<a href="#">INET_PORT_RESERVATION_TOKEN</a>	Contains a port reservation token for a block of TCP or UDP ports.
<a href="#">ip_mreq</a>	Provides multicast group information for IPv4 addresses.
<a href="#">ip_mreq_source</a>	Provides multicast group and source information for IPv4 addresses.
<a href="#">ip_msfilter</a>	Provides multicast filtering parameters for IPv4 addresses.
<a href="#">ipv6_mreq</a>	Provides multicast group information for IPv6 addresses.
<a href="#">linger</a>	Maintains information about a specific socket that specifies how that socket should behave when data is queued to be sent and the <b>closesocket</b> function is called on the socket.
<a href="#">NAPI_DOMAIN_DESCRIPTION_BLOB</a>	Describes a domain handled by a namespace provider for the NS_EMAIL namespace.
<a href="#">NAPI_PROVIDER_INSTALLATION_BLOB</a>	Contains the information required to install a namespace provider for the NS_EMAIL namespace.
<a href="#">NS_SERVICE_INFO</a>	Contains information about a network service or a network service type in the context of a specified namespace, or a set of default namespaces.
<a href="#">PROTOCOL_INFO</a>	Contains information about a protocol.
<a href="#">protoent</a>	Contains the name and protocol numbers that correspond to a given protocol name.

STRUCTURE	DESCRIPTION
<a href="#">REAL_TIME_NOTIFICATION_SETTING_INPUT</a>	Contains input settings to apply for the <a href="#">REAL_TIME_NOTIFICATION_CAPABILITY</a> transport setting for a TCP socket that is used with <a href="#">ControlChannelTrigger</a> to receive background network notifications in a Windows Store app
<a href="#">REAL_TIME_NOTIFICATION_SETTING_OUTPUT</a>	Contains the output settings from a query for the <a href="#">REAL_TIME_NOTIFICATION_CAPABILITY</a> transport setting for a TCP socket that is used with <a href="#">ControlChannelTrigger</a> to receive background network notifications in a Windows Store app.
<a href="#">RIO_EXTENSION_FUNCTION_TABLE</a>	Contains information on the functions that implement the Winsock registered I/O extensions.
<a href="#">RIO_BUF</a>	Specifies a portion of a registered buffer used for sending or receiving network data with the Winsock registered I/O extensions.
<a href="#">RIO_BUFFERID</a>	Specifies a registered buffer descriptor used with the Winsock registered I/O extensions.
<a href="#">RIO_CQ</a>	Specifies a completion queue descriptor used for I/O completion notification by send and receive requests with the Winsock registered I/O extensions.
<a href="#">RIO_NOTIFICATION_COMPLETION</a>	Specifies the method for I/O completion to be used with a <a href="#">RIONotify</a> function for sending or receiving network data with the Winsock registered I/O extensions.
<a href="#">RIO_RQ</a>	Specifies a socket descriptor used by send and receive requests with the Winsock registered I/O extensions.
<a href="#">RIORESULT</a>	Contains data used to indicate request completion results used with the Winsock registered I/O extensions.
<a href="#">RM_FEC_INFO</a>	Specifies settings for using forward error correction (FEC) with Reliable Multicast.
<a href="#">RM_RECEIVER_STATS</a>	Provides statistical information for a Reliable Multicast receiver session.
<a href="#">RM_SEND_WINDOW</a>	Specifies the Reliable Multicast send window.
<a href="#">RM_SENDER_STATS</a>	Provides statistical information for a Reliable Multicast sender session.
<a href="#">servent</a>	Stores or returns the name and service number for a given service name.
<a href="#">SERVICE_ADDRESS</a>	Contains address information for a service.
<a href="#">SERVICE_ADDRESSES</a>	Contains an array of <a href="#">SERVICE_ADDRESS</a> data structures.

STRUCTURE	DESCRIPTION
<a href="#">SERVICE_INFO</a>	Contains information about a network service or a network service type.
<a href="#">SERVICE_TYPE_INFO_ABS</a>	Contains information about a network service type.
<a href="#">SERVICE_TYPE_VALUE_ABS</a>	Contains information about a network-service type value. The information may be specific to a namespace.
<a href="#">sockaddr</a>	Contains socket address information. The <a href="#">sockaddr</a> structure varies depending on the protocol selected. Limited to IPv4; use <a href="#">SOCKADDR_STORAGE</a> instead.
<a href="#">SOCKADDR_IRDA</a>	Used with IrDA socket operations, defined by address family <a href="#">AF_IRDA</a> .
<a href="#">SOCKADDR_STORAGE</a>	Stores socket address information, and is sufficiently large to store IPv4 or IPv6 address information, promoting protocol-family and protocol-version independence. Use this structure in place of the <a href="#">sockaddr</a> structure.
<a href="#">SOCKET_ADDRESS</a>	Stores protocol-specific address information.
<a href="#">SOCKET_ADDRESS_LIST</a>	Stores an array of <a href="#">SOCKET_ADDRESS</a> structures that contain protocol-specific address information.
<a href="#">SOCKET_PEER_TARGET_NAME</a>	Contains the IP address and name for a peer target and the type of security protocol to be used on a socket.
<a href="#">SOCKET_SECURITY_QUERY_INFO</a>	Contains security information returned by the <a href="#">WSAQuerySocketSecurity</a> function.
<a href="#">SOCKET_SECURITY_QUERY_TEMPLATE</a>	Contains the security template used by the <a href="#">WSAQuerySocketSecurity</a> function.
<a href="#">SOCKET_SECURITY_SETTINGS</a>	Specifies generic security requirements for a socket.
<a href="#">SOCKET_SECURITY_SETTINGS_IPSEC</a>	Specifies various security requirements and settings that are specific to IPsec.
<a href="#">timeval</a>	Used to specify time values. Associated with the Berkeley Software Distribution (BSD) file Time.h.
<a href="#">TRANSMIT_FILE_BUFFERS</a>	Specifies data to be transmitted before and after file data during a <a href="#">TransmitFile</a> transfer operation.
<a href="#">TRANSMIT_PACKETS_ELEMENT</a>	Specifies a single data element to be transmitted by the <a href="#">TransmitPackets</a> function.
<a href="#">TRANSPORT_SETTING_ID</a>	Specifies the transport setting ID used by the <a href="#">SIO_APPLY_TRANSPORT_SETTING</a> and <a href="#">SIO_QUERY_TRANSPORT_SETTING</a> IOCTLs to apply or query the transport setting for a socket.
<a href="#">WSABUF</a>	Enables the creation or manipulation of a data buffer.

STRUCTURE	DESCRIPTION
<a href="#">WSACOMPLETION</a>	Specifies completion notification settings for I/O control calls made to a registered namespace.
<a href="#">WSADATA</a>	Contains information about the Windows Sockets implementation.
<a href="#">WSAMSG</a>	Stores address and optional control information about connected and unconnected sockets. Used with the <a href="#">LPFN_WSARECVMSG (WSARecvMsg)</a> function.
<a href="#">WSANAMESPACE_INFO</a>	Contains registration information for a namespace provider.
<a href="#">WSANAMESPACE_INFOEX</a>	Contains enhanced registration information for a namespace provider.
<a href="#">WSANETWORKEVENTS</a>	Stores a socket's internal information about network events.
<a href="#">WSANSCLASSINFO</a>	Provides individual parameter information for a specific Winsock namespace.
<a href="#">WSAOVERLAPPED</a>	Provides a communication medium between the initiation of an overlapped I/O operation and its subsequent completion.
<a href="#">WSAPOLLFD</a>	Stores socket information used by the <a href="#">WSAPoll</a> function.
<a href="#">WSAPROTOCOL_INFO</a>	Stores or retrieves complete information for a given protocol.
<a href="#">WSAPROTOCOL_INFOW</a>	Store or retrieves complete information for a given protocol. The protocol name is represented as an array of Unicode characters.
<a href="#">WSAPROTOCOLCHAIN</a>	Contains a counted list of Catalog Entry identifiers that comprise a protocol chain.
<a href="#">WSAQUERYSET</a>	Provides relevant information about a given service.
<a href="#">WSAQUERYSET2</a>	Provides relevant information about a given service.
<a href="#">WSASERVICECLASSINFO</a>	Contains information about a specified service class.
<a href="#">WSAVERSION</a>	Provides version comparison in Winsock.

# RIO\_BUFFERID

3/5/2021 • 2 minutes to read • [Edit Online](#)

The RIO\_BUFFERID typedef specifies a registered buffer descriptor used with the Winsock registered I/O extensions.

```
typedef struct RIO_BUFFERID_t* RIO_BUFFERID, **PRIO_BUFFERID;
```

## RIO\_BUFFERID

A data type that specifies a registered buffer descriptor used with send and receive requests.

## Remarks

The Winsock registered I/O extensions operate primarily on registered buffers using RIO\_BUFFERID objects. An application obtains a RIO\_BUFFERID for an existing buffer using the [RIORegisterBuffer](#) function. An application can release a registration using the [RIODeregisterBuffer](#) function.

When an existing buffer is registered as a RIO\_BUFFERID object using the [RIORegisterBuffer](#) function, certain internal resources are allocated from physical memory, and the existing application buffer will be locked into physical memory. The [RIODeregisterBuffer](#) function is called to deregister the buffer, free these internal resources, and allow the buffer to be unlocked and released from physical memory.

Repeated registration and deregistration of application buffers using the Winsock registered I/O extensions may cause significant performance degradation. The following buffer management approaches should be considered when designing an application using the Winsock registered I/O extensions to minimize repeated registration and deregistration of application buffers:

- Maximize the reuse of buffers.
- Maintain a limited pool of unused registered buffers for use by the application.
- Maintain a limited pool of registered buffers and perform buffer copies between these registered buffers and other unregistered buffers.

The RIO\_BUFFERID typedef is defined in the *Mswsockdef.h* header file which is automatically included in the *Mswsock.h* header file. The *Mswsockdef.h* header file should never be used directly.

## Requirements

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	Mswsockdef.h (include Mswsock.h)

## See also

**RIO\_BUF**

**RIODeregisterBuffer**

**RIOReceive**

**RIOReceiveEx**

**RIORegisterBuffer**

**RIOSend**

**RIOSendEx**

# RIO\_CQ

3/5/2021 • 3 minutes to read • [Edit Online](#)

The RIO\_CQ typedef specifies a completion queue descriptor used for I/O completion notification by send and receive requests with the Winsock registered I/O extensions.

```
typedef struct RIO_CQ_t* RIO_CQ, **PRIO_CQ;
```

## RIO\_CQ

A data type that specifies a completion queue descriptor used for I/O completion notification by send and receive requests.

## Remarks

The RIO\_CQ object is used for I/O completion notification of send and receive networking requests by the Winsock registered I/O extensions.

An application can use the [RIONotify](#) function to request notification when a RIO\_CQ completion queue is not empty. An application can also poll the status at any time of a RIO\_CQ completion queue in a non-blocking way using the [RIODequeueCompletion](#) function.

The RIO\_CQ object is created using the [RIOCreateCompletionQueue](#) function. At creation time, the application must specify the size of the queue, which determines how many completion entries it can hold. When an application calls the [RIOCreateRequestQueue](#) function to obtain a RIO\_RQ handle, the application must specify a RIO\_CQ handle for send completions and a RIO\_CQ handle for receive completions. These handles may be identical when the same queue should be used for send and receive completion. The [RIOCreateRequestQueue](#) function also requires a maximum number of outstanding send and receive operations, which are charged against the capacity of the associated completion queue or queues. If the queues do not have sufficient capacity remaining, the [RIOCreateRequestQueue](#) call will fail with [WSAENOBUFS](#).

The notification behavior for a completion queue is set when the RIO\_CQ is created.

For a completion queue that uses an event, the Type member of the [RIO\\_NOTIFICATION\\_COMPLETION](#) structure is set to [RIO\\_EVENT\\_COMPLETION](#). The Event.EventHandle member should contain the handle for an event created by the [WSACreateEvent](#) or [CreateEvent](#) function. To receive the [RIONotify](#) completion, the application should wait on the specified event handle using [WSAWaitForMultipleEvents](#) or a similar wait routine. If the application plans to reset and reuse the event, the application can reduce overhead by setting the Event.NotifyReset member to a non-zero value. This causes the event to be automatically reset by the [RIONotify](#) function when the notification occurs. This mitigates the need to call the [WSAResetEvent](#) function to reset the event between calls to the [RIONotify](#) function.

For a completion queue that uses an I/O completion port, the Type member of the [RIO\\_NOTIFICATION\\_COMPLETION](#) structure is set to [RIO\\_IOCP\\_COMPLETION](#). The locp.locpHandle member should contain the handle for an I/O completion port created by the [CreateIoCompletionPort](#) function. To receive the [RIONotify](#) completion, the application should call the [GetQueuedCompletionStatus](#) or [GetQueuedCompletionStatusEx](#) function. The application should provide a dedicated [OVERLAPPED](#) object for the completion queue, and it may also use the locp.CompletionKey member to distinguish [RIONotify](#) requests on the completion queue from other I/O completions including [RIONotify](#) completions for other completion queues.

#### NOTE

For purposes of efficiency, access to the completion queues (**RIO\_CQ** structs) and request queues (**RIO\_RQ** structs) are not protected by synchronization primitives. If you need to access a completion or request queue from multiple threads, access should be coordinated by a critical section, slim reader write lock or similar mechanism. This locking is not needed for access by a single thread. Different threads can access separate requests/completion queues without locks. The need for synchronization occurs only when multiple threads try to access the same queue. Synchronization is also required if multiple threads issue sends and receives on the same socket because the send and receive operations use the socket's request queue.

If multiple threads attempt to access the same **RIO\_CQ** using [RIODequeueCompletion](#), access must be coordinated by a critical section, slim reader writer lock , or similar mutual exclusion mechanism. If the completion queues are not shared, mutual exclusion is not required.

When a completion queue is no longer needed, an application can close it using the [RIOCcloseCompletionQueue](#) function.

The **RIO\_CQ** typedef is defined in the *Mswsockdef.h* header file which is automatically included in the *Mswsock.h* header file. The *Mswsockdef.h* header file should never be used directly.

## Thread Safety

If multiple threads attempt to access the same **RIO\_CQ** using [RIODequeueCompletion](#), access must be coordinated by a critical section, slim reader writer lock , or similar mutual exclusion mechanism. If the completion queues are not shared, mutual exclusion is not required.

## Requirements

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	Mswsockdef.h (include Mswsock.h)

## See also

[CreateIoCompletionPort](#)

[CreateEvent](#)

[GetQueuedCompletionStatus](#)

[GetQueuedCompletionStatusEx](#)

[OVERLAPPED](#)

[RIO\\_NOTIFICATION\\_COMPLETION](#)

[RIO\\_NOTIFICATION\\_COMPLETION\\_TYPE](#)

[RIO\\_RQ](#)

[RIOCloseCompletionQueue](#)

[RIOCreateCompletionQueue](#)

[RIOCreateRequestQueue](#)

[RIODequeueCompletion](#)

[RIONotify](#)

[WSACreateEvent](#)

[WSAResetEvent](#)

[WSAWaitForMultipleEvents](#)

# RIO\_RQ

3/5/2021 • 2 minutes to read • [Edit Online](#)

The **RIO\_RQ** typedef specifies a socket descriptor used by send and receive requests with the Winsock registered I/O extensions.

```
typedef struct RIO_RQ_t* RIO_RQ, **PRIO_RQ;
```

## RIO\_RQ

A data type that specifies a socket descriptor used by send and receive requests.

## Remarks

The Winsock registered I/O extensions operate primarily on a **RIO\_RQ** object rather than a socket. An application obtains a **RIO\_RQ** for an existing socket using the [RIOCreateRequestQueue](#) function. The input socket must have been created by calling the [WSASocket](#) function with the **WSA\_FLAG\_RIO** flag set in the *dwFlags* parameter.

After obtaining a **RIO\_RQ** object, the underlying socket descriptor remains valid. An application may continue to use the underlying socket to set and query socket options, issue IOCTLs and ultimately close the socket.

### NOTE

For purposes of efficiency, access to the completion queues (**RIO\_CQ** structs) and request queues (**RIO\_RQ** structs) are not protected by synchronization primitives. If you need to access a completion or request queue from multiple threads, access should be coordinated by a critical section, slim reader write lock or similar mechanism. This locking is not needed for access by a single thread. Different threads can access separate requests/completion queues without locks. The need for synchronization occurs only when multiple threads try to access the same queue. Synchronization is also required if multiple threads issue sends and receives on the same socket because the send and receive operations use the socket's request queue.

The **RIO\_RQ** typedef is defined in the *Mswsockdef.h* header file which is automatically included in the *Mswsock.h* header file. The *Mswsockdef.h* header file should never be used directly.

## Requirements

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	Mswsockdef.h (include Mswsock.h)

## See also

[RIOCreateRequestQueue](#)

[RIOReceive](#)

[RIOReceiveEx](#)

[RIOResizeRequestQueue](#)

[RIOSend](#)

[RIOSendEx](#)

[WSASocket](#)

# sockaddr

3/5/2021 • 2 minutes to read • [Edit Online](#)

The sockaddr structure varies depending on the protocol selected. Except for the *sin\*\_family* parameter, sockaddr contents are expressed in network byte order.

Winsock functions using sockaddr are not strictly interpreted to be pointers to a sockaddr structure. The 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 **SOCKADDR\_STORAGE** structure also stores socket address information and the structure is sufficiently large to store IPv4 or IPv6 address information. The use of the **SOCKADDR\_STORAGE** structure promotes protocol-family and protocol-version independence, and simplifies development. It is recommended that the **SOCKADDR\_STORAGE** structure be used in place of the sockaddr structure. The **SOCKADDR\_STORAGE** structure is supported on Windows Server 2003 and later.

The sockaddr structure and sockaddr\_in structures below are used with IPv4. Other protocols use similar structures.

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

struct sockaddr_in {
    short  sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char   sin_zero[8];
};
```

The sockaddr\_in6 and sockaddr\_in6\_old structures below are used with IPv6.

```
struct sockaddr_in6 {
    short  sin6_family;
    u_short sin6_port;
    u_long  sin6_flowinfo;
    struct in6_addr sin6_addr;
    u_long  sin6_scope_id;
};

typedef struct sockaddr_in6 SOCKADDR_IN6;
typedef struct sockaddr_in6 *PSOCKADDR_IN6;
typedef struct sockaddr_in6 FAR *LPSOCKADDR_IN6;

struct sockaddr_in6_old {
    short  sin6_family;
    u_short sin6_port;
    u_long  sin6_flowinfo;
    struct in6_addr sin6_addr;
};
```

On the Microsoft Windows Software Development Kit (SDK) released for Windows Vista and later, **SOCKADDR** and **SOCKADDR\_IN** typedef tags are defined for sockaddr and sockaddr\_in structures as follows:

```

typedef struct sockaddr {
#if (_WIN32_WINNT < 0x0600)
    u_short sa_family;
#else
    ADDRESS_FAMILY sa_family;
#endif //(_WIN32_WINNT < 0x0600)
    CHAR sa_data[14];
} SOCKADDR, *PSOCKADDR, FAR *LPSOCKADDR;

typedef struct sockaddr_in {
#if(_WIN32_WINNT < 0x0600)
    short sin_family;
#else //(_WIN32_WINNT < 0x0600)
    ADDRESS_FAMILY sin_family;
#endif //(_WIN32_WINNT < 0x0600)
    USHORT sin_port;
    IN_ADDR sin_addr;
    CHAR sin_zero[8];
} SOCKADDR_IN, *PSOCKADDR_IN;

```

On the Windows SDK released for Windows Vista and later, the organization of header files has changed and the `sockaddr` and `sockaddr_in` structures are defined in the `Ws2def.h` header file, not the `Winsock2.h` header file. The `Ws2def.h` header file is automatically included by the `Winsock2.h` header file. The `sockaddr_in6` structure is defined in the `Ws2ipdef.h` header file, not the `Ws2tcpip.h` header file. The `Ws2ipdef.h` header file is automatically included by the `Ws2tcpip.h` header file. The `Ws2def.h` and `Ws2ipdef.h` header files should never be used directly.

## Example Code

The following example demonstrates the use of the `sockaddr` structure.

```

// Declare variables
SOCKET ListenSocket;
struct sockaddr_in saServer;
hostent* localHost;
char* localIP;

// Create a listening socket
ListenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

// Get the local host information
localHost = gethostbyname("");
localIP = inet_ntoa ((*((struct in_addr *)localHost->h_addr_list)));

// Set up the sockaddr structure
saServer.sin_family = AF_INET;
saServer.sin_addr.s_addr = inet_addr(localIP);
saServer.sin_port = htons(5150);

// Bind the listening socket using the
// information in the sockaddr structure
bind( ListenSocket,(SOCKADDR*) &saServer, sizeof(saServer) );

```

## See Also

[SOCKADDR\\_STORAGE](#)



# Winsock Tracing Events

3/5/2021 • 2 minutes to read • [Edit Online](#)

This section describes detailed information on specific Winsock Tracing Events details.

Winsock tracing is a troubleshooting feature that can be enabled in retail binaries to trace certain Windows socket events with minimal overhead. This feature allows for better diagnostic capabilities for developers and product support. Winsock network event tracing supports tracing socket operations for IPv4 and IPv6 applications. Winsock catalog change tracing supports tracing changes made to the Winsock catalog by layered service providers (LSPs).

## NOTE

Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use [Windows Filtering Platform](#).

Winsock tracing uses Event Tracing for Windows (ETW), a general-purpose, high-speed tracing facility provided by the operating system. ETW provides a tracing mechanism for events raised by both user-mode applications and kernel-mode device drivers. ETW can enable and disable logging dynamically, making it easy to perform detailed tracing in production environments without requiring reboots or application restarts. Support for Winsock tracing using ETW was added on Windows Vista and later. For general information on ETW, see [Improve Debugging And Performance Tuning With ETW](#).

The following list provides detailed information for each Winsock tracing event. For additional information on any event, click the event name.

EVENT NAME	DESCRIPTION
<a href="#">AFD_EVENT_CREATE</a>	Winsock network tracing event for a socket creation operation.
<a href="#">AFD_EVENT_CLOSE</a>	Winsock network tracing event for socket close operation.
<a href="#">WINSOCK_WS2HELP_LSP_INSTALL</a>	Winsock catalog change event for a layered service provider (LSP) installation operation.
<a href="#">WINSOCK_WS2HELP_LSP_REMOVE</a>	Winsock catalog change event for a layered service provider (LSP) removal operation.
<a href="#">WINSOCK_WS2HELP_LSP_DISABLE</a>	Winsock catalog change event for a layered service provider (LSP) disable operation.
<a href="#">WINSOCK_WS2HELP_LSP_RESET</a>	Winsock catalog change event for a Winsock catalog reset operation.

## Related topics

## [Improve Debugging And Performance Tuning With ETW](#)

[Winsock Tracing](#)

[Winsock Tracing Levels](#)

[Control of Winsock Tracing](#)

[Winsock Network Event Tracing Details](#)

[Winsock Catalog Change Tracing Details](#)

# AFD\_EVENT\_CREATE event

3/5/2021 • 5 minutes to read • [Edit Online](#)

The AFD\_EVENT\_CREATE event is a Winsock network tracing event for a socket creation operation.

```
const EVENT_DESCRIPTOR AFD_EVENT_CREATE = {0x3e8, 0x0, 0x10, 0x4, 0xa, 0x3e8, 0x8000000000000004};
```

## Parameters

### *EnterExit*

Information on this event.

The following table lists the possible values for the *EnterExit* parameter:

VALUE	MEANING
0	The start of a Winsock request.
1	The Winsock request completed.
2	The Winsock AFD driver took an internal action (aborting a connection, for example).
3	The TCP/IP driver caused this event to occur. This usually indicates a data notification.
4	The Winsock AFD driver caused this event to occur (setting a socket option, for example).

### *Location*

A private field used internally.

### *Process*

The [EPROCESS](#) address of the process that owns the related socket. This is an opaque structure that serves as the process object for a process. For more information, see the Windows Driver Kit documentation for the [EPROCESS](#) structure.

### *Endpoint*

The AFD\_ENDPOINT address of the socket.

### *AddressFamily*

The address family specification for the socket. Possible values for the address family are defined in the *Ws2def.h* header file. Note that the *Ws2def.h* header file is automatically included in *Winsock2.h*, and should never be used directly.

The values currently supported are AF\_INET or AF\_INET6, which are the Internet address family formats for IPv4 and IPv6. Other options for address family (AF\_NETBIOS for use with NetBIOS, for example) are supported if a Windows Sockets service provider for the address family is installed.

The table below lists common values for address family although many other values are possible.

AF	MEANING
AF_UNSPEC 0	The address family is unspecified.
AF_INET 2	The Internet Protocol version 4 (IPv4) address family.
AF_IPX 6	The IPX/SPX address family. This address family is only supported if the NWLink IPX/SPX NetBIOS Compatible Transport protocol is installed. This address family is not supported on Windows Vista and later.
AF_APPLETALK 16	The AppleTalk address family. This address family is only supported if the AppleTalk protocol is installed. This address family is not supported on Windows Vista and later.
AF_NETBIOS 17	The NetBIOS address family. This address family is only supported if the Windows Sockets provider for NetBIOS is installed. The Windows Sockets provider for NetBIOS is supported on 32-bit versions of Windows. This provider is installed by default on 32-bit versions of Windows. The Windows Sockets provider for NetBIOS is not supported on 64-bit versions of windows. The Windows Sockets provider for NetBIOS only supports sockets where the <i>type</i> parameter is set to <b>SOCK_DGRAM</b> . The Windows Sockets provider for NetBIOS is not directly related to the <a href="#">NetBIOS</a> programming interface. The NetBIOS programming interface is not supported on Windows Vista, Windows Server 2008, and later.
AF_INET6 23	The Internet Protocol version 6 (IPv6) address family.
AF_IRDA 26	The Infrared Data Association (IrDA) address family. This address family is only supported if the computer has an infrared port and driver installed.
AF_BTH 32	The Bluetooth address family. This address family is only supported if the computer has a Bluetooth adapter and driver installed.

### *SocketType*

The type specification for the new socket. Possible values for the socket type are defined in the *Winsock2.h* header file.

The following table lists the possible values for the *type* parameter supported for Windows Sockets 2:

TYPE	MEANING
SOCK_STREAM 1	A socket type that provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. This socket type uses the Transmission Control Protocol (TCP) for the Internet address family (AF_INET or AF_INET6).
SOCK_DGRAM 2	A socket type that supports datagrams, which are connectionless, unreliable buffers of a fixed (typically small) maximum length. This socket type uses the User Datagram Protocol (UDP) for the Internet address family (AF_INET or AF_INET6).
SOCK_RAW 3	A socket type that provides a raw socket that allows an application to manipulate the next upper-layer protocol header. To manipulate the IPv4 header, the <a href="#">IP_HDRINCL</a> socket option must be set on the socket. To manipulate the IPv6 header, the <a href="#">IPV6_HDRINCL</a> socket option must be set on the socket.
SOCK_RDM 4	A socket type that provides a reliable message datagram. An example of this type is the Pragmatic General Multicast (PGM) multicast protocol implementation in Windows, often referred to as <a href="#">reliable multicast programming</a> . This <i>type</i> value is only supported if the Reliable Multicast Protocol is installed.
SOCK_SEQPACKET 5	A socket type that provides a pseudo-stream packet based on datagrams.

#### Protocol

The protocol to be used. The possible options for the *protocol* parameter are specific to the address family and socket type specified. Possible values for the *protocol* are defined in the *Wsrm.h* header file and the **IPPROTO** enumeration type defined in the *Ws2def.h* header file. Note that the *Ws2def.h* header file is automatically included in *Winsock2.h*, and should never be used directly.

If a value of 0 is specified, the caller does not wish to specify a protocol and the service provider will choose the *protocol* to use.

The table below lists common values for the *protocol* although many other values are possible.

PROTOCOL	MEANING
IPPROTO_ICMP 1	The Internet Control Message Protocol (ICMP). This is a possible value when the <i>af</i> parameter is <b>AF_UNSPEC</b> , <b>AF_INET</b> , or <b>AF_INET6</b> and the <i>type</i> parameter is <b>SOCK_RAW</b> or unspecified.
IPPROTO_IGMP 2	The Internet Group Management Protocol (IGMP). This is a possible value when the <i>af</i> parameter is <b>AF_UNSPEC</b> , <b>AF_INET</b> , or <b>AF_INET6</b> and the <i>type</i> parameter is <b>SOCK_RAW</b> or unspecified.

PROTOCOL	MEANING
BTHPROTO_RFCOMM 3	The Bluetooth Radio Frequency Communications (Bluetooth RFCOMM) protocol. This is a possible value when the <i>af</i> parameter is AF_BTH and the <i>type</i> parameter is SOCK_STREAM.
IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the <i>af</i> parameter is AF_INET or AF_INET6 and the <i>type</i> parameter is SOCK_STREAM.
IPPROTO_UDP 17	The User Datagram Protocol (UDP). This is a possible value when the <i>af</i> parameter is AF_INET or AF_INET6 and the <i>type</i> parameter is SOCK_DGRAM.
IPPROTO_ICMPV6 58	The Internet Control Message Protocol Version 6 (ICMPv6). This is a possible value when the <i>af</i> parameter is AF_UNSPEC, AF_INET, or AF_INET6 and the <i>type</i> parameter is SOCK_RAW or unspecified.
IPPROTO_RM 113	The PGM protocol for reliable multicast. This is a possible value when the <i>af</i> parameter is AF_INET and the <i>type</i> parameter is SOCK_RDM. This protocol is also called IPPROTO_PGM. This <i>protocol</i> value is only supported if the Reliable Multicast Protocol is installed.

#### ProcessId

The actual process ID or an indicator if the event was a result of Winsock code run in a system process or in a deferred procedure call (DPC) context (contexts outside the user process).

#### Status

The NTSTATUS code for the operation.

## Remarks

The AFD\_EVENT\_CREATE event is traced for a Winsock network operation to create a socket. The channel for this event is Winsock-AFD. The level for this event is informational.

## Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]

## See also

[Control of Winsock Tracing](#)

[EVENT\\_DESCRIPTOR](#)

[Winsock Tracing](#)

[Winsock Tracing Levels](#)

[Winsock Catalog Change Tracing Details](#)

# AFD\_EVENT\_CLOSE event

3/5/2021 • 2 minutes to read • [Edit Online](#)

The AFD\_EVENT\_CLOSE event is a Winsock network tracing event for socket close operation.

```
const EVENT_DESCRIPTOR AFD_EVENT_CLOSE = {0x3e9, 0x0, 0x10, 0x4, 0xf, 0x3e9, 0x8000000000000004};
```

## Parameters

### *EnterExit*

Information on this event.

The following table lists the possible values for the *EnterExit* parameter:

VALUE	MEANING
0	The start of a Winsock request.
1	The Winsock request completed.
2	The Winsock AFD driver took an internal action (aborting a connection, for example).
3	The TCP/IP driver caused this event to occur. This usually indicates a data notification.
4	The Winsock AFD driver caused this event to occur (setting a socket option, for example).

### *Location*

A private field used internally.

### *Process*

The [EPROCESS](#) address of the process that owns the related socket. This is an opaque structure that serves as the process object for a process. For more information, see the Windows Driver Kit documentation for the [EPROCESS](#) structure.

### *Endpoint*

The AFD\_ENDPOINT address of the socket.

### *Status*

The NTSTATUS code for the operation.

## Remarks

The AFD\_EVENT\_CLOSE event is traced for a Winsock network operation to close a socket. The channel for this event is Winsock-AFD. The level for this event is informational.

## Requirements

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]

## See also

[Control of Winsock Tracing](#)

[EVENT\\_DESCRIPTOR](#)

[Winsock Tracing](#)

[Winsock Tracing Levels](#)

[Winsock Catalog Change Tracing Details](#)

# WINSOCK\_WS2HELP\_LSP\_INSTALL event

3/5/2021 • 2 minutes to read • [Edit Online](#)

## NOTE

Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use [Windows Filtering Platform](#).

The **WINSOCK\_WS2HELP\_LSP\_INSTALL** event is a Winsock catalog change event for a layered service provider (LSP) installation operation.

```
const EVENT_DESCRIPTOR WINSOCK_WS2HELP_LSP_INSTALL = {0x1, 0x0, 0x10, 0x0, 0x0, 0x0, 0x0, 0x8000000000000000};
```

## Parameters

### *LSP Name*

The name of the LSP as obtained from the **szProtocol** member of the [WSAPROTOCOL\\_INFO](#) structure for the LSP being installed.

### *Catalog*

The Winsock catalog (32-bit or 64-bit) where the LSP is being installed. This is an integer value that is either 32 or 64.

### *Installer*

The module filename of the application making the LSP install call.

### *GUID*

The GUID value of the Winsock transport provider that the LSP is being installed under.

### *Category*

The **dwCatalogEntryId** member of the [WSAPROTOCOL\\_INFO](#) structure for the LSP being installed.

## Remarks

The **WINSOCK\_WS2HELP\_LSP\_INSTALL** event is traced for an LSP install operation when a protocol entry is installed into the Winsock catalog.

## Requirements

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

## See also

[Control of Winsock Tracing](#)

[Winsock Tracing](#)

[Winsock Tracing Levels](#)

[Winsock Catalog Change Tracing Details](#)

# WINSOCK\_WS2HELP\_LSP\_REMOVE event

3/5/2021 • 2 minutes to read • [Edit Online](#)

## NOTE

Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use [Windows Filtering Platform](#).

The **WINSOCK\_WS2HELP\_LSP\_REMOVE** event is a Winsock catalog change event for a layered service provider (LSP) removal operation.

```
const EVENT_DESCRIPTOR WINSOCK_WS2HELP_LSP_REMOVE = {0x2, 0x0, 0x10, 0x0, 0x0, 0x0, 0x0, 0x8000000000000000};
```

## Parameters

### *LSP Name*

The name of the LSP as obtained from the **szProtocol** member of the [WSAPROTOCOL\\_INFO](#) structure for the LSP being removed.

### *Catalog*

The Winsock catalog (32-bit or 64-bit) where the LSP is being removed. This is an integer value that is either 32 or 64.

### *Installer*

The module filename of the application making the LSP remove call.

### *GUID*

The GUID value of the Winsock transport provider that the LSP is being removed from.

### *Category*

The **dwCatalogEntryId** member of the [WSAPROTOCOL\\_INFO](#) structure for the LSP being removed.

## Remarks

The **WINSOCK\_WS2HELP\_LSP\_REMOVE** event is traced for an LSP removal operation when a protocol entry is removed from the Winsock catalog.

## Requirements

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

## See also

[Control of Winsock Tracing](#)

[Winsock Tracing](#)

[Winsock Tracing Levels](#)

[Winsock Catalog Change Tracing Details](#)

# WINSOCK\_WS2HELP\_LSP\_DISABLE event

3/5/2021 • 2 minutes to read • [Edit Online](#)

## NOTE

Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use [Windows Filtering Platform](#).

The **WINSOCK\_WS2HELP\_LSP\_DISABLE** event is a Winsock catalog change event for a layered service provider (LSP) disable operation.

```
const EVENT_DESCRIPTOR WINSOCK_WS2HELP_LSP_DISABLE = {0x3, 0x0, 0x10, 0x0, 0x0, 0x0, 0x0, 0x8000000000000000};
```

## Parameters

### *LSP Name*

The name of the LSP as obtained from the **szProtocol** member of the [WSAPROTOCOL\\_INFO](#) structure for the LSP being disabled.

### *Catalog*

The Winsock catalog (32-bit or 64-bit) where the LSP is being disabled. This is an integer value that is either 32 or 64.

### *Installer*

The module filename of the application making the LSP disable call.

### *GUID*

The GUID value of the Winsock transport provider that the LSP is being disabled.

### *Category*

The **dwCatalogEntryId** member of the [WSAPROTOCOL\\_INFO](#) structure for the LSP being disabled.

This event has no parameters.

## Remarks

The **WINSOCK\_WS2HELP\_LSP\_DISABLE** event is traced for an LSP disable operation when a protocol entry is disabled in the Winsock catalog.

## Requirements

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]

Requirement	Value
Minimum supported server	Windows Server 2008 [desktop apps only]

## See also

[Control of Winsock Tracing](#)

[Winsock Tracing](#)

[Winsock Tracing Levels](#)

[Winsock Catalog Change Tracing Details](#)

# WINSOCK\_WS2HELP\_LSP\_RESET event

3/5/2021 • 2 minutes to read • [Edit Online](#)

## NOTE

Layered Service Providers are deprecated. Starting with Windows 8 and Windows Server 2012, use [Windows Filtering Platform](#).

The **WINSOCK\_WS2HELP\_LSP\_RESET** event is a Winsock catalog change event for a Winsock catalog reset operation.

```
const EVENT_DESCRIPTOR WINSOCK_WS2HELP_LSP_RESET = {0x4, 0x0, 0x10, 0x0, 0x0, 0x0, 0x0, 0x8000000000000000};
```

## Parameters

### *Catalog*

The Winsock catalog (32-bit or 64-bit) that is being reset. This is an integer value that is either 32 or 64.

## Remarks

The **WINSOCK\_WS2HELP\_LSP\_RESET** event is traced for an Winsock Layered Service Provider (LSP) operation when the Winsock catalog is reset.

## Requirements

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]

## See also

[Control of Winsock Tracing](#)

[Winsock Tracing](#)

[Winsock Tracing Levels](#)

[Winsock Catalog Change Tracing Details](#)

# Winsock SPI

3/5/2021 • 2 minutes to read • [Edit Online](#)

The Winsock Service Provider Interface, or Winsock SPI, is a specialized discipline of Winsock used to create providers; transport providers such as TCP/IP or IPX/SPX protocol stacks use the Winsock SPI, as do namespace providers such as the Internet's Domain Naming System (DNS).

Traditional network programming, such as enabling applications to communicate over the network, does not require use of Winsock SPI interfaces; use standard [Winsock](#) interfaces instead.

The Winsock SPI uses the following function prefix naming convention.

PREFIX	MEANING	DESCRIPTION
WSP	Windows Sockets Service Provider	Transport service provider entry points
WPU	Windows Sockets Provider Upcall	Ws2_32.dll entry points for service providers
WSC	Windows Sockets Configuration	WS2_32.dll entry points for installation applets
NSP	Namespace Provider	Namespace provider entry points

# Winsock SPI Functions

3/5/2021 • 7 minutes to read • [Edit Online](#)

The following list provides concise descriptions of each Winsock SPI function. For additional information on any Winsock SPI function, click the function name. For information about prefix conventions, see [Winsock SPI](#). For information about the Winsock SPI usage, see [About the Winsock SPI](#).

WINSOCK SPI FUNCTION	DESCRIPTION
<a href="#">NSPCleanup</a>	Terminates the use of a Winsock namespace service provider.
<a href="#">NSPGetServiceClassInfo</a>	Retrieves class information (schema) pertaining to a specified namespace provider.
<a href="#">NSPIInstallServiceClass</a>	Registers service class schema within the namespace providers.
<a href="#">NSPIoctl</a>	Sends an IOCTL to a namespace service provider.
<a href="#">NSPLookupServiceBegin</a>	Initiates a client query of a namespace version 1 service provider that is constrained by information contained within a <a href="#">WSAQUERYSET</a> structure.
<a href="#">NSPLookupServiceEnd</a>	Frees a handle after previous calls to <a href="#">NSPLookupServiceBegin</a> and <a href="#">NSPLookupServiceNext</a> .
<a href="#">NSPLookupServiceNext</a>	Retrieves requested information from a namespace version 1 service provider. This function is called only after obtaining a handle from a previous call to <a href="#">NSPLookupServiceBegin</a> .
<a href="#">NSPRemoveServiceClass</a>	Permanently removes a specified service class from the namespace.
<a href="#">NSPSetService</a>	Registers or deregisters a service instance within a namespace.
<a href="#">NSPStartup</a>	Retrieves dynamic information about a provider, such as the list of the DLL entry points.
<a href="#">NSPv2Cleanup</a>	Notifies a namespace service provider version 2 (NSPv2) provider that a client session has terminated.
<a href="#">NSPv2ClientSessionRundown</a>	Notifies a NSPv2 provider that the client session is terminating.
<a href="#">NSPv2LookupServiceBegin</a>	Initiates a client query of NSPv2 service provider that is constrained by the information contained within a <a href="#">WSAQUERYSET2</a> structure.
<a href="#">NSPv2LookupServiceEnd</a>	Frees a handle after previous calls to <a href="#">NSPv2LookupServiceBegin</a> and <a href="#">NSPv2LookupServiceNextEx</a> .

WINSOCK SPI FUNCTION	DESCRIPTION
<a href="#">NSPv2LookupServiceNextEx</a>	Retrieves the requested information from a NSPv2 provider. This function is called only after obtaining a handle from a previous call to <a href="#">NSPv2LookupServiceBegin</a> .
<a href="#">NSPv2SetServiceEx</a>	Registers or deregisters a name or service instance within a namespace of a NSPv2 provider.
<a href="#">NSPv2Startup</a>	Notifies a NSPv2 provider that a new client process is to begin using the provider.
<a href="#">WPUCloseEvent</a>	Closes an open event object handle.
<a href="#">WPUCloseSocketHandle</a>	Closes an existing socket handle.
<a href="#">WPUCloseThread</a>	Closes a thread opened with a call to <a href="#">WPUOpenCurrentThread</a> .
<a href="#">WPUCOMPLETEOVERLAPPEDREQUEST</a>	Performs overlapped I/O completion notification for overlapped I/O operations.
<a href="#">WPUCREATEEVENT</a>	Creates a new event object.
<a href="#">WPUCREATESOCKETHANDLE</a>	Creates a new socket handle.
<a href="#">WPUDFISSET</a>	Checks the membership of the specified socket handle.
<a href="#">WPUGETPROVIDERPATH</a>	Retrieves the DLL path for the specified provider.
<a href="#">WPUMODIFYIFSHANDLE</a>	Receives a possibly modified IFS handle from Ws2_32.dll.
<a href="#">WPUOPENCURRENTTHREAD</a>	Opens a handle to the current thread that can be used with overlapped functions in a layered service provider.
<a href="#">WPUPOSTMESSAGE</a>	Performs the standard Windows <a href="#">PostMessage</a> function in a way that maintains backward compatibility with older versions of WSOCK32.dll.
<a href="#">WPUQUERYBLOCKINGCALLBACK</a>	Returns a pointer to a callback function the service provider should invoke periodically while servicing blocking operations.
<a href="#">WPUQUERYSOCKETHANDLECONTEXT</a>	Queries the context value associated with the specified socket handle.
<a href="#">WPUQUEUEAPC</a>	Queues a user mode-asynchronous procedure call (APC) to the specified thread in order to facilitate invocation of overlapped I/O completion routines.
<a href="#">WPURESETEVENT</a>	Resets the state of the specified event object to nonsignaled.
<a href="#">WPUSETEVENT</a>	Sets the state of the specified event object to signaled.

WINSOCK SPI FUNCTION	DESCRIPTION
<a href="#">WSAAdvertiseProvider</a>	Makes a specific namespace version 2 provider available for all eligible clients.
<a href="#">WSAProviderCompleteAsyncCall</a>	Notifies a client when an asynchronous call to a namespace version 2 provider is completed.
<a href="#">WSAUnadvertiseProvider</a>	Makes a specific namespace version 2 provider no longer available for clients.
<a href="#">WSCDeinstallProvider</a>	Removes the specified transport service provider from the system configuration database.
<a href="#">WSCDeinstallProvider32</a>	Removes the specified 32-bit transport provider from the system configuration database on a 64-bit platform.
<a href="#">WSCEnableNSProvider</a>	Enables or disables a specified namespace provider.
<a href="#">WSCEnableNSProvider32</a>	Enables or disables a specified 32-bit namespace provider on a 64-bit platform.
<a href="#">WSCEnumNameSpaceProviders32</a>	Returns information on available 32-bit namespace providers on a 64-bit platform.
<a href="#">WSCEnumNameSpaceProvidersEx32</a>	Returns information on available 32-bit namespace providers on a 64-bit platform.
<a href="#">WSCEnumProtocols</a>	Retrieves information about available transport protocols.
<a href="#">WSCEnumProtocols32</a>	Retrieves information about available transport protocols in the 32-bit catalog on 64-bit platforms.
<a href="#">WSCGetApplicationCategory</a>	Retrieves the layered service provider (LSP) categories associated with an application.
<a href="#">WSCGetProviderInfo</a>	Retrieves the data associated with an information class for a layered service provider.
<a href="#">WSCGetProviderInfo32</a>	Retrieves the data associated with an information class for a 32-bit layered service provider on a 64-bit platform.
<a href="#">WSCGetProviderPath</a>	Retrieves the DLL path for the specified provider.
<a href="#">WSCGetProviderPath32</a>	Retrieves the DLL path for the specified 32-bit provider on a 64-bit platform.
<a href="#">WSCInstallNameSpace</a>	Installs a namespace provider.
<a href="#">WSCInstallNameSpace32</a>	Installs a 32-bit namespace provider on a 64 bit platform.
<a href="#">WSCInstallNameSpaceEx</a>	Installs a namespace provider.
<a href="#">WSCInstallNameSpaceEx32</a>	Installs a 32-bit namespace provider on a 64 bit platform.

WINSOCK SPI FUNCTION	DESCRIPTION
<a href="#">WSCInstallProvider</a>	Installs a transport service provider into the system configuration database.
<a href="#">WSCInstallProvider64_32</a>	Installs the specified transport service provider into the 32-bit and 64-bit system configuration databases on a 64-bit platform.
<a href="#">WSCInstallProviderAndChains</a>	Installs a 32-bit transport service provider as well as its specific protocol chains into the system configuration database on a 32-bit platform.
<a href="#">WSCInstallProviderAndChains64_32</a>	Installs the specified transport provider and its specific protocol chains into both the 32-bit and 64-bit system configuration databases on a 64-bit platform.
<a href="#">WSCSetApplicationCategory</a>	Sets the permitted layered service provider categories associated with an application.
<a href="#">WSCSetProviderInfo</a>	Sets the data value for the specified information class for a layered service provider.
<a href="#">WSCSetProviderInfo32</a>	Sets the data value for the specified information class for a 32-bit layered service provider on a 64-bit platform.
<a href="#">WSCUnInstallNameSpace</a>	Uninstalls a namespace provider.
<a href="#">WSCUnInstallNameSpace32</a>	Uninstalls a specific 32-bit namespace provider on a 32-bit platform.
<a href="#">WSCUpdateProvider</a>	Modifies a transport service provider in the system configuration database.
<a href="#">WSCUpdateProvider32</a>	Modifies a 32-bit transport service provider in the system configuration database on a 64-bit platform.
<a href="#">WSCWriteNameSpaceOrder</a>	Changes the order of available namespace providers.
<a href="#">WSCWriteNameSpaceOrder32</a>	Changes the order of available Windows Sockets (Winsock) 2 namespace providers in a 32-bit catalog on a 64-bit platform.
<a href="#">WSCWriteProviderOrder</a>	Re-orders available transport service providers.
<a href="#">WSCWriteProviderOrder32</a>	Re-orders available transport service providers in a 32-bit catalog on a 64-bit platform.
<a href="#">WSPAccept</a>	Conditionally accepts a connection based on the return value of a condition function.
<a href="#">WSPAddressToString</a>	Converts all components of a <a href="#">sockaddr</a> structure into a human readable–numeric string representation of the address.

WINSOCK SPI FUNCTION	DESCRIPTION
<a href="#">WSPAsyncSelect</a>	Requests Windows message-based event notification of network events for a socket.
<a href="#">WSPBind</a>	Associates a local address with a socket.
<a href="#">WSPCancelBlockingCall</a>	Cancels a blocking call that is currently in progress.
<a href="#">WSPCleanup</a>	Terminates use of the Winsock service provider.
<a href="#">WSPCloseSocket</a>	Closes a socket.
<a href="#">WSPConnect</a>	Establishes a connection to a peer, exchanges connect data, and specifies needed quality of service based on the supplied flow specification.
<a href="#">WSPDuplicateSocket</a>	Returns a <a href="#">WSAPROTOCOL_INFO</a> structure that can be used to create a new socket descriptor for a shared socket.
<a href="#">WSPEnumNetworkEvents</a>	Reports occurrences of network events for the indicated socket.
<a href="#">WSPEventSelect</a>	Specifies an event object to be associated with the supplied set of network events.
<a href="#">WSPGetOverlappedResult</a>	Returns the results of an overlapped operation on the specified socket.
<a href="#">WSPGetPeerName</a>	Retrieves the address of the peer to which a socket is connected.
<a href="#">WSPGetQOSByName</a>	Initializes a <a href="#">QOS</a> structure based on a named template, or retrieves an enumeration of the available template names.
<a href="#">WSPGetSockName</a>	Retrieves the local name for a socket.
<a href="#">WSPGetSockOpt</a>	Retrieves a socket option.
<a href="#">WSPIoctl</a>	Controls the mode of a socket.
<a href="#">WSPJoinLeaf</a>	Joins a leaf node into a multipoint session, exchanges connect data, and specifies needed quality of service based on the supplied flow specifications.
<a href="#">WSPListen</a>	Establishes a socket to listen for incoming connections.
<a href="#">WSPRecv</a>	Receives data on a socket.
<a href="#">WSPRecvDisconnect</a>	Terminates reception on a socket, and if the socket is connection oriented, retrieves the disconnect data.
<a href="#">WSPRecvFrom</a>	Receives a datagram and stores the source address.

WINSOCK SPI FUNCTION	DESCRIPTION
<a href="#">WSPSelect</a>	Determines the status of one or more sockets.
<a href="#">WSPSend</a>	Sends data on a connected socket.
<a href="#">WSPSendDisconnect</a>	Initiates termination of the connection for a socket and sends disconnect data.
<a href="#">WSPSendTo</a>	Sends data to a specific destination using overlapped I/O.
<a href="#">WSPSetSockOpt</a>	Sets a socket option.
<a href="#">WSPShutdown</a>	Disables sends and/or receives on a socket.
<a href="#">WSPSocket</a>	Creates a socket.
<a href="#">WSPStartup</a>	Initiates use of a Winsock service provider by a client.
<a href="#">WSPStringToAddress</a>	Converts a human-readable numeric string to a socket address structure suitable for passing to Winsock interface that take such a structure.

## 32-bit SPI Functions for 64-bit Platforms

### NOTE

These calls are strictly 32-bit versions of native WSC function calls for use on 64-bit platforms by 32-bit processes. The definitions and semantics of these specific 32-bit calls are the same as their native counterparts.

32-BIT WINSOCK SPI FUNCTION	DESCRIPTION
<a href="#">WSCDeinstallProvider32</a>	Removes the specified 32-bit transport service provider from the system configuration database on a 64-bit platform.
<a href="#">WSCEnableNSProvider32</a>	Changes the state of a given 32-bit namespace provider on a 64-bit platform.
<a href="#">WSCEnumNameSpaceProviders32</a>	Returns a set of available 32-bit namespace providers on a 64-bit platform.
<a href="#">WSCEnumNameSpaceProvidersEx32</a>	Returns information on available 32-bit namespace providers on a 64-bit platform.
<a href="#">WSCEnumProtocols32</a>	Retrieves information about available 32-bit transport protocols on a 64-bit platform.
<a href="#">WSCGetProviderInfo32</a>	Retrieves the data associated with an information class for a 32-bit layered service provider on a 64-bit platform.

32-BIT WINSOCK SPI FUNCTION	DESCRIPTION
<a href="#">WSCGetProviderPath32</a>	Retrieves the DLL path for the specified 32-bit provider on a 64-bit platform.
<a href="#">WSCInstallNameSpace32</a>	Installs a 32-bit namespace provider on a 64-bit platform.
<a href="#">WSCInstallNameSpaceEx32</a>	Installs a namespace provider.
<a href="#">WSCInstallProvider64_32</a>	Installs a transport service provider into the 32-bit and 64-bit system configuration databases on a 64-bit platform.
<a href="#">WSCInstallProviderAndChains64_32</a>	Installs a transport service provider and its specific protocol chains into both the 32-bit and 64-bit system configuration databases on a 64-bit platform.
<a href="#">WSCSetProviderInfo32</a>	Sets the data value for specified information class for a layered service provider on a 64-bit platform.
<a href="#">WSCUnInstallNameSpace32</a>	Uninstalls a 32-bit namespace provider on a 64-bit platform.
<a href="#">WSCUpdateProvider32</a>	Modifies a 32-bit transport service provider in the system configuration database on a 64-bit platform.
<a href="#">WSCWriteNameSpaceOrder32</a>	Modifies the order of available namespace providers in a 32-bit catalog on a 64-bit platform.

# Winsock SPI Structures

3/5/2021 • 2 minutes to read • [Edit Online](#)

The following list provides concise descriptions of each Winsock SPI structure. For additional information on any Winsock SPI structure, click the structure name.

WINSOCK SPI STRUCTURE	DESCRIPTION
<a href="#">NSP_ROUTINE</a>	Contains information on the functions implemented by a namespace service provider version 1 (NSPv1) provider.
<a href="#">NSPV2_ROUTINE</a>	Contains information on the functions implemented by a namespace service provider version 2 (NSPv2) provider.
<a href="#">WSATHREADID</a>	Enables a provider to identify a thread on which asynchronous procedure calls (APCs) can be queued.
<a href="#">WSC_PROVIDER_AUDIT_INFO</a>	Contains audit information for a layered service provider (LSP) entry.
<a href="#">WSPDATA</a>	Contains service provider information.
<a href="#">WSPPROC_TABLE</a>	Contains a table of pointers to service provider functions.
<a href="#">WSPUPCALLTABLE</a>	Contains a table of pointers to service provider upcall functions.

# Windows Sockets Error Codes

3/22/2021 • 16 minutes to read • [Edit Online](#)

Most Windows Sockets 2 functions do not return the specific cause of an error when the function returns. For information, see the [Handling Winsock Errors](#) topic.

The [WSAGetLastError](#) function returns the last error that occurred for the calling thread. When a particular Windows Sockets function indicates an error has occurred, this function should be called immediately to retrieve the extended error code for the failing function call. These error codes and a short text description associated with an error code are defined in the *Winerror.h* header file. The [FormatMessage](#) function can be used to obtain the message string for the returned error.

For information on how to handle error codes when porting socket applications to Winsock, see [Error Codes - errno, h\\_errno and WSAGetLastError](#).

The following list describes the possible error codes returned by the [WSAGetLastError](#) function. Errors are listed in numerical order with the error macro name. Some error codes defined in the *Winsock2.h* header file are not returned from any function.

RETURN CODE/VALUE	DESCRIPTION
<b>WSA_INVALID_HANDLE</b> 6	Specified event object handle is invalid. An application attempts to use an event object, but the specified handle is not valid.
<b>WSA_NOT_ENOUGH_MEMORY</b> 8	Insufficient memory available. An application used a Windows Sockets function that directly maps to a Windows function. The Windows function is indicating a lack of required memory resources.
<b>WSA_INVALID_PARAMETER</b> 87	One or more parameters are invalid. An application used a Windows Sockets function which directly maps to a Windows function. The Windows function is indicating a problem with one or more parameters.
<b>WSA_OPERATION_ABORTED</b> 995	Overlapped operation aborted. An overlapped operation was canceled due to the closure of the socket, or the execution of the SIO_FLUSH command in <a href="#">WSAIoctl</a> .
<b>WSA_IO_INCOMPLETE</b> 996	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 <a href="#">WSAGetOverlappedResult</a> (with the <i>fWait</i> flag set to FALSE) in a polling mode to determine when an overlapped operation has completed, get this error code until the operation is complete.

RETURN CODE/VALUE	DESCRIPTION
<b>WSA_IO_PENDING</b> 997	<p>Overlapped operations will complete later.</p> <p>The application has initiated an overlapped operation that cannot be completed immediately. A completion indication will be given later when the operation has been completed.</p>
<b>WSAEINTR</b> 10004	<p>Interrupted function call.</p> <p>A blocking operation was interrupted by a call to <a href="#">WSACancelBlockingCall</a>.</p>
<b>WSAEBADF</b> 10009	<p>File handle is not valid.</p> <p>The file handle supplied is not valid.</p>
<b>WSAEACCES</b> 10013	<p>Permission denied.</p> <p>An attempt was made to access a socket in a way forbidden by its access permissions. An example is using a broadcast address for <a href="#">sendto</a> without broadcast permission being set using <a href="#">setsockopt(SO_BROADCAST)</a>.</p> <p>Another possible reason for the WSAEACCES error is that when the <a href="#">bind</a> function is called (on Windows NT 4.0 with SP4 and later), another application, service, or kernel mode driver is bound to the same address with exclusive access. Such exclusive access is a new feature of Windows NT 4.0 with SP4 and later, and is implemented by using the <a href="#">SO_EXCLUSIVEADDRUSE</a> option.</p>
<b>WSAEFAULT</b> 10014	<p>Bad address.</p> <p>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 <a href="#">sockaddr</a> structure, is smaller than the sizeof(sockaddr).</p>
<b>WSAEINVAL</b> 10022	<p>Invalid argument.</p> <p>Some invalid argument was supplied (for example, specifying an invalid level to the <a href="#">setsockopt</a> function). In some instances, it also refers to the current state of the socket—for instance, calling <a href="#">accept</a> on a socket that is not listening.</p>
<b>WSAEMFILE</b> 10024	<p>Too many open files.</p> <p>Too many open sockets. Each implementation may have a maximum number of socket handles available, either globally, per process, or per thread.</p>

RETURN CODE/VALUE	DESCRIPTION
WSAEWOULDBLOCK 10035	<p>Resource temporarily unavailable.</p> <p>This error is returned from operations on nonblocking sockets that cannot be completed immediately, for example <a href="#">recv</a> when no data is queued to be read from the socket. It is a nonfatal error, and the operation should be retried later. It is normal for WSAEWOULDBLOCK to be reported as the result from calling <a href="#">connect</a> on a nonblocking SOCK_STREAM socket, since some time must elapse for the connection to be established.</p>
WSAEINPROGRESS 10036	<p>Operation now in progress.</p> <p>A blocking operation is currently executing. Windows Sockets only allows a single blocking operation—per-task or thread—to be outstanding, 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.</p>
WSAEALREADY 10037	<p>Operation already in progress.</p> <p>An operation was attempted on a nonblocking socket with an operation already in progress—that is, calling <a href="#">connect</a> a second time on a nonblocking socket that is already connecting, or canceling an asynchronous request (<a href="#">WSAAAsyncGetXbyY</a>) that has already been canceled or completed.</p>
WSAENOTSOCK 10038	<p>Socket operation on nonsocket.</p> <p>An operation was attempted on something that is not a socket. Either the socket handle parameter did not reference a valid socket, or for <a href="#">select</a>, a member of an <b>fd_set</b> was not valid.</p>
WSAEDESTADDRREQ 10039	<p>Destination address required.</p> <p>A required address was omitted from an operation on a socket. For example, this error is returned if <a href="#">sendto</a> is called with the remote address of ADDR_ANY.</p>
WSAEMSGSIZE 10040	<p>Message too long.</p> <p>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 was smaller than the datagram itself.</p>
WSAEPROTOTYPE 10041	<p>Protocol wrong type for socket.</p> <p>A protocol was specified in the <a href="#">socket</a> 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.</p>

RETURN CODE/VALUE	DESCRIPTION
<b>WSAENOPROTOOPT</b> 10042	Bad protocol option. An unknown, invalid or unsupported option or level was specified in a <a href="#">getsockopt</a> or <a href="#">setsockopt</a> call.
<b>WSAEPROTONOSUPPORT</b> 10043	Protocol not supported. The requested protocol has not been configured into the system, or no implementation for it exists. For example, a <a href="#">socket</a> call requests a SOCK_DGRAM socket, but specifies a stream protocol.
<b>WSAESOCKTNOSUPPORT</b> 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 <a href="#">socket</a> call, and the implementation does not support SOCK_RAW sockets at all.
<b>WSAEOPNOTSUPP</b> 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 is trying to accept a connection on a datagram socket.
<b>WSAEPFNOSUPPORT</b> 10046	Protocol family not supported. The protocol family has not been configured into the system or no implementation for it exists. This message has a slightly different meaning from WSAEAFNOSUPPORT. However, it is interchangeable in most cases, and all Windows Sockets functions that return one of these messages also specify WSAEAFNOSUPPORT.
<b>WSAEAFNOSUPPORT</b> 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 (that is, AF_INET for Internet Protocols) and a generic protocol type (that is, SOCK_STREAM). This error is returned if an incorrect protocol is explicitly requested in the <a href="#">socket</a> call, or if an address of the wrong family is used for a socket, for example, in <a href="#">sendto</a> .

RETURN CODE/VALUE	DESCRIPTION
<b>WSAEADDRINUSE</b> 10048	<p>Address already in use.</p> <p>Typically, only one usage of each socket address (protocol/IP address/port) is permitted. This error occurs if an application attempts to <b>bind</b> a socket to an IP address/port that has already been used for an existing socket, or a socket that was not closed properly, or one that is still in the process of closing. For server applications that need to <b>bind</b> multiple sockets to the same port number, consider using <b>setsockopt</b> (SO_REUSEADDR). Client applications usually need not call <b>bind</b> at all—<b>connect</b> chooses an unused port automatically. When <b>bind</b> 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 another function later, including <b>connect</b>, <b>listen</b>, <b>WSAConnect</b>, or <b>WSAJoinLeaf</b>.</p>
<b>WSAEADDRNOTAVAIL</b> 10049	<p>Cannot assign requested address.</p> <p>The requested address is not valid in its context. This normally results from an attempt to <b>bind</b> to an address that is not valid for the local computer. This can also result from <b>connect</b>, <b>sendto</b>, <b>WSAConnect</b>, <b>WSAJoinLeaf</b>, or <b>WSASendTo</b> when the remote address or port is not valid for a remote computer (for example, address or port 0).</p>
<b>WSAENETDOWN</b> 10050	<p>Network is down.</p> <p>A socket operation encountered a dead network. This could indicate a serious failure of the network system (that is, the protocol stack that the Windows Sockets DLL runs over), the network interface, or the local network itself.</p>
<b>WSAENETUNREACH</b> 10051	<p>Network is unreachable.</p> <p>A socket operation was attempted to an unreachable network. This usually means the local software knows no route to reach the remote host.</p>
<b>WSAENETRESET</b> 10052	<p>Network dropped connection on reset.</p> <p>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 <b>setsockopt</b> if an attempt is made to set <b>SO_KEEPALIVE</b> on a connection that has already failed.</p>
<b>WSAECONNABORTED</b> 10053	<p>Software caused connection abort.</p> <p>An established connection was aborted by the software in your host computer, possibly due to a data transmission time-out or protocol error.</p>

RETURN CODE/VALUE	DESCRIPTION
WSAECONNRESET 10054	<p>Connection reset by peer.</p> <p>An 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, the host or remote network interface is disabled, or the remote host uses a hard close (see <a href="#">setsockopt</a> 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.</p>
WSAENOBUFS 10055	<p>No buffer space available.</p> <p>An operation on a socket could not be performed because the system lacked sufficient buffer space or because a queue was full.</p>
WSAEISCONN 10056	<p>Socket is already connected.</p> <p>A connect request was made on an already-connected socket. Some implementations also return this error if <a href="#">sendto</a> is called on a connected SOCK_DGRAM socket (for SOCK_STREAM sockets, the <i>to</i> parameter in <a href="#">sendto</a> is ignored) although other implementations treat this as a legal occurrence.</p>
WSAENOTCONN 10057	<p>Socket is not connected.</p> <p>A request to send or receive data was disallowed because the socket is not connected and (when sending on a datagram socket using <a href="#">sendto</a>) no address was supplied. Any other type of operation might also return this error—for example, <a href="#">setsockopt</a> setting <a href="#">SO_KEEPALIVE</a> if the connection has been reset.</p>
WSAESHUTDOWN 10058	<p>Cannot send after socket shutdown.</p> <p>A request to send or receive data was disallowed because the socket had already been shut down in that direction with a previous <a href="#">shutdown</a> call. By calling <a href="#">shutdown</a> a partial close of a socket is requested, which is a signal that sending or receiving, or both have been discontinued.</p>
WSAETOOMANYREFS 10059	<p>Too many references.</p> <p>Too many references to some kernel object.</p>

RETURN CODE/VALUE	DESCRIPTION
<b>WSAETIMEDOUT</b> 10060	Connection timed out. A connection attempt failed because the connected party did not properly respond after a period of time, or the established connection failed because the connected host has failed to respond.
<b>WSAECONNREFUSED</b> 10061	Connection refused. No connection could be made because the target computer actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host—that is, one with no server application running.
<b>WSAELOOP</b> 10062	Cannot translate name. Cannot translate a name.
<b>WSAENAMETOOLONG</b> 10063	Name too long. A name component or a name was too long.
<b>WSAEHOSTDOWN</b> 10064	Host is down. A socket operation failed because the destination host is 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.
<b>WSAEHOSTUNREACH</b> 10065	No route to host. A socket operation was attempted to an unreachable host. See WSAENETUNREACH.
<b>WSAENOTEMPTY</b> 10066	Directory not empty. Cannot remove a directory that is not empty.
<b>WSAEPROCLIM</b> 10067	Too many processes. A Windows Sockets implementation may have a limit on the number of applications that can use it simultaneously. <a href="#">WSAStartup</a> may fail with this error if the limit has been reached.
<b>WSAEUSERS</b> 10068	User quota exceeded. Ran out of user quota.
<b>WSAEDQUOT</b> 10069	Disk quota exceeded. Ran out of disk quota.

RETURN CODE/VALUE	DESCRIPTION
<b>WSAESTALE</b> 10070	Stale file handle reference. The file handle reference is no longer available.
<b>WSAEREMOTE</b> 10071	Item is remote. The item is not available locally.
<b>WSASYSNOTREADY</b> 10091	<p>Network subsystem is unavailable.</p> <p>This error is returned by <a href="#">WSAStartup</a> 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:</p> <ul style="list-style-type: none"> <li>• That the appropriate Windows Sockets DLL file is in the current path.</li> <li>• 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.</li> <li>• The Windows Sockets implementation documentation to be sure all necessary components are currently installed and configured correctly.</li> </ul>
<b>WSAVERNOTSUPPORTED</b> 10092	<p>Winsock.dll version out of range.</p> <p>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.</p>
<b>WSANOTINITIALISED</b> 10093	<p>Successful WSAStartup not yet performed.</p> <p>Either the application has not called <a href="#">WSAStartup</a> or <a href="#">WSAStartup</a> failed. The application may be accessing a socket that the current active task does not own (that is, trying to share a socket between tasks), or <a href="#">WSACleanup</a> has been called too many times.</p>
<b>WSAEDISCON</b> 10101	<p>Graceful shutdown in progress.</p> <p>Returned by <a href="#">WSARecv</a> and <a href="#">WSARecvFrom</a> to indicate that the remote party has initiated a graceful shutdown sequence.</p>
<b>WSAENOMORE</b> 10102	<p>No more results.</p> <p>No more results can be returned by the <a href="#">WSALookupServiceNext</a> function.</p>

RETURN CODE/VALUE	DESCRIPTION
<b>WSAECANCELLED</b> 10103	Call has been canceled. A call to the <a href="#">WSALookupServiceEnd</a> function was made while this call was still processing. The call has been canceled.
<b>WSAEINVALIDPROCTABLE</b> 10104	Procedure call table is invalid. The service provider procedure call table is invalid. A service provider returned a bogus procedure table to Ws2_32.dll. This is usually caused by one or more of the function pointers being <b>NULL</b> .
<b>WSAEINVALIDPROVIDER</b> 10105	Service provider is invalid. The requested service provider is invalid. This error is returned by the <a href="#">WSCGetProviderInfo</a> and <a href="#">WSCGetProviderInfo32</a> functions if the protocol entry specified could not be found. This error is also returned if the service provider returned a version number other than 2.0.
<b>WSAEPROVIDERFAILEDINIT</b> 10106	Service provider failed to initialize. The requested service provider could not be loaded or initialized. This error is returned if either a service provider's DLL could not be loaded ( <a href="#">LoadLibrary</a> failed) or the provider's <a href="#">WSPStartup</a> or <a href="#">NSPStartup</a> function failed.
<b>WSASYSCALLFAILURE</b> 10107	System call failure. A system call that should never fail has failed. This is a generic error code, returned under various conditions. Returned when a system call that should never fail does fail. For example, if a call to <a href="#">WaitForMultipleEvents</a> fails or one of the registry functions fails trying to manipulate the protocol/namespace catalogs. Returned when a provider does not return SUCCESS and does not provide an extended error code. Can indicate a service provider implementation error.
<b>WSASERVICE_NOT_FOUND</b> 10108	Service not found. No such service is known. The service cannot be found in the specified name space.
<b>WSATYPE_NOT_FOUND</b> 10109	Class type not found. The specified class was not found.
<b>WSA_E_NO_MORE</b> 10110	No more results. No more results can be returned by the <a href="#">WSALookupServiceNext</a> function.

RETURN CODE/VALUE	DESCRIPTION
<b>WSA_E_CANCELLED</b> 10111	Call was canceled. A call to the <a href="#">WSALookupServiceEnd</a> function was made while this call was still processing. The call has been canceled.
<b>WSAEREFUSED</b> 10112	Database query was refused. A database query failed because it was actively refused.
<b>WSAHOST_NOT_FOUND</b> 11001	Host not found. No such host is known. The name is not an official host name 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 that the specified name could not be found in the relevant database.
<b>WSATRY AGAIN</b> 11002	Nonauthoritative host not found. This is usually a temporary error during host name 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.
<b>WSANO_RECOVERY</b> 11003	This is a nonrecoverable error. This indicates that some sort of nonrecoverable error occurred during a database lookup. This may be because the database files (for example, BSD-compatible HOSTS, SERVICES, or PROTOCOLS files) could not be found, or a DNS request was returned by the server with a severe error.
<b>WSANO_DATA</b> 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 host name-to-address translation attempt (using <a href="#">gethostbyname</a> or <a href="#">WSAAAsyncGetHostByName</a> ) which uses the DNS (Domain Name Server). An MX record is returned but no A record—indicating the host itself exists, but is not directly reachable.
<b>WSA_QOS_RECEIVERS</b> 11005	QoS receivers. At least one QoS reserve has arrived.
<b>WSA_QOS_SENDERS</b> 11006	QoS senders. At least one QoS send path has arrived.
<b>WSA_QOS_NO_SENDERS</b> 11007	No QoS senders. There are no QoS senders.

RETURN CODE/VALUE	DESCRIPTION
<b>WSA_QOS_NO_RECEIVERS</b> 11008	QoS no receivers. There are no QoS receivers.
<b>WSA_QOS_REQUEST_CONFIRMED</b> 11009	QoS request confirmed. The QoS reserve request has been confirmed.
<b>WSA_QOS_ADMISSION_FAILURE</b> 11010	QoS admission error. A QoS error occurred due to lack of resources.
<b>WSA_QOS_POLICY_FAILURE</b> 11011	QoS policy failure. The QoS request was rejected because the policy system couldn't allocate the requested resource within the existing policy.
<b>WSA_QOS_BAD_STYLE</b> 11012	QoS bad style. An unknown or conflicting QoS style was encountered.
<b>WSA_QOS_BAD_OBJECT</b> 11013	QoS bad object. A problem was encountered with some part of the filterspec or the provider-specific buffer in general.
<b>WSA_QOS_TRAFFIC_CTRL_ERROR</b> 11014	QoS traffic control error. An error with the underlying traffic control (TC) API as the generic QoS request was converted for local enforcement by the TC API. This could be due to an out of memory error or to an internal QoS provider error.
<b>WSA_QOS_GENERIC_ERROR</b> 11015	QoS generic error. A general QoS error.
<b>WSA_QOS_ESERVICETYPE</b> 11016	QoS service type error. An invalid or unrecognized service type was found in the QoS flowspec.
<b>WSA_QOS_EFLOWSPEC</b> 11017	QoS flowspec error. An invalid or inconsistent flowspec was found in the <b>QOS</b> structure.
<b>WSA_QOS_EPROVSPECBUF</b> 11018	Invalid QoS provider buffer. An invalid QoS provider-specific buffer.

RETURN CODE/VALUE	DESCRIPTION
<b>WSA_QOS_EFILTERSTYLE</b> 11019	Invalid QoS filter style. An invalid QoS filter style was used.
<b>WSA_QOS_EFILTERTYPE</b> 11020	Invalid QoS filter type. An invalid QoS filter type was used.
<b>WSA_QOS_EFILTERCOUNT</b> 11021	Incorrect QoS filter count. An incorrect number of QoS FILTERSPECs were specified in the FLOWDESCRIPTOR.
<b>WSA_QOS_EOBJLENGTH</b> 11022	Invalid QoS object length. An object with an invalid ObjectLength field was specified in the QoS provider-specific buffer.
<b>WSA_QOS_EFLOWCOUNT</b> 11023	Incorrect QoS flow count. An incorrect number of flow descriptors was specified in the QoS structure.
<b>WSA_QOS_EUNKOWNPSOBJ</b> 11024	Unrecognized QoS object. An unrecognized object was found in the QoS provider-specific buffer.
<b>WSA_QOS_EPOLICYOBJ</b> 11025	Invalid QoS policy object. An invalid policy object was found in the QoS provider-specific buffer.
<b>WSA_QOS_EFLOWDESC</b> 11026	Invalid QoS flow descriptor. An invalid QoS flow descriptor was found in the flow descriptor list.
<b>WSA_QOS_EPSFLOWSPEC</b> 11027	Invalid QoS provider-specific flowspec. An invalid or inconsistent flowspec was found in the QoS provider-specific buffer.
<b>WSA_QOS_EPSFILTERSPEC</b> 11028	Invalid QoS provider-specific filterspec. An invalid FILTERSPEC was found in the QoS provider-specific buffer.
<b>WSA_QOS_ESDMODEOBJ</b> 11029	Invalid QoS shape discard mode object. An invalid shape discard mode object was found in the QoS provider-specific buffer.

RETURN CODE/VALUE	DESCRIPTION
<b>WSA_QOS_ESHAPERATEOBJ</b> 11030	Invalid QoS shaping rate object. An invalid shaping rate object was found in the QoS provider-specific buffer.
<b>WSA_QOS_RESERVED_PETYPE</b> 11031	Reserved policy QoS element type. A reserved policy element was found in the QoS provider-specific buffer.

## Requirements

REQUIREMENT	VALUE
Header	Winsock2.h; Winerror.h

## See also

[Error Codes - errno, h\\_errno and WSAGetLastError](#)

[Handling Winsock Errors](#)

[FormatMessage](#)

[WSAGetLastError](#)